Updating the Vertex Separation of a Dynamically Changing Tree

by

Peter Olsar

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

Abstract

This thesis presents several algorithms that update the vertex separation of a tree after the tree is modified; the vertex separation of a graph measures the largest number of vertices to the left of and including a vertex that are adjacent to vertices to the right of the vertex, when the vertices in the graph are arranged in the best possible linear ordering. Vertex separation was introduced by Lipton and Tarjan and has since been applied mainly in VLSI design. The tree is modified by either attaching another tree or removing a subtree. The first algorithm handles the special case when another tree is attached to the root, and the second algorithm updates the vertex separation after a subtree of the root is removed. The last two algorithms solve the more general problem when subtrees are attached to or removed from arbitrary vertices; they have good running time performance only in the amortized sense. The running time of all our algorithms is sublinear in the number of vertices in the tree, assuming certain information is precomputed for the tree. This improves upon current algorithms by Skodinis and Ellis, Sudborough, and Turner, both of which have linear running time for this problem. Lower and upper bounds on the vertex separation of a general graph are also derived. Furthermore, analogous bounds are presented for the cutwidth of a general graph, where the cutwidth of a graph equals the maximum number of edges that cross over a vertex, when the vertices in the graph are arranged in the best possible linear ordering.

Acknowledgements

First of all, I would like to thank my supervisors, Naomi Nishimura and Prabhakar Ragde, who have given me plenty of valuable advice, support, and encouragement throughout the course of my research and writing of this thesis. I am also thankful to my readers, Daniel Brown and Ming Li, who agreed to review the thesis and suggested improvements. I am indebted to Jason Hinek for his help with LaTeX and Unix.

I would also like to thank my friends who have contributed positively to my experience as a graduate student at the University of Waterloo. Jason Hinek for stimulating conversations in the office and games of tennis. Yongtao Hu for being such a great friend. Jeff Farrar for teaching me a lot about history. Ethan Toombs and Anmar Khadra for the weekly squash games. Alex Tumanov and Bohdan Krushelnytskyy for many insightful discussions. Finally, all residents of Minota Hagey Residence, my home for two years, have my thanks for creating such a great community.

for my mother and step-father

Contents

Chapter 1

Introduction

The objective of a graph layout problem [DPS02] is to find a linear ordering, or layout, of the input graph's vertices in a way that optimizes a certain objective function. The decision versions of many layout problems are NP-complete on general graphs, but become tractable on special classes of graphs, such as trees. This thesis is mainly concerned with this class of graphs.

We look at two types of layout problems: the minimum vertex separation problem and the minimum cutwidth problem. The minimum vertex separation problem was introduced as a tool for finding good planar graph separations [LT79]; a graph separation is a partition of the graph's vertices into several sets such that a certain objective function, such as the number of edges with endpoints in different sets, is minimized. Further applications of the minimum vertex separation problem occur in algorithms for VLSI design [Lei80, Möh90] and in complexity theory [CS76]. The minimum cutwidth problem was first used as a model for the minimum number of channels required to lay out a circuit on a line [AH73]. It has since been applied in information retrieval [Bot93], automatic graph drawing [Mut95], and network reliability [Kar00].

Informally, the vertex separation of a layout is the maximum number of vertices to the left of or including a vertex that are adjacent to vertices to the right of that vertex. The cutwidth of the layout is the maximum number of edges such that one endpoint is to the left of a vertex or is equal to that vertex, and the other endpoint is to the right of the vertex. These concepts will be defined rigorously in Chapter

2. The goal is to find a layout, called an optimal layout, that minimizes either vertex separation or cutwidth. The vertex separation or cutwidth of a graph is this minimum value.

The minimum vertex separation and cutwidth problems, especially the minimum vertex separation problem, turn out to be related quite closely to several other well-known problems in theoretical computer science. Kinnersley showed that the vertex separation and pathwidth of a graph are equivalent concepts [Kin92]. Pathwidth, a notion similar to treewidth, is an important metric in the theory of graph minors [RS85]. Although pathwidth is both a shorter and more usual way of referring to the concept, we will keep using the term 'vertex separation,' because the results presented in this thesis are better described and understood by using the original definition of vertex separation.

The edge search number problem on a graph [Par76], which involves finding the minimum number of guards required to capture a mobile fugitive hiding in an edge of the graph, is also closely related to the concept of vertex separation [EST94]; the edge search number of a graph is greater than or equal to the vertex separation of the graph, and it is not greater than the vertex separation plus two. The cutwidth and edge search number of a graph are equal if the maximum degree of a vertex in the graph is at most three [MS89]. The related node search number problem on a graph was shown to be equal to the vertex separation of the graph plus one [KP86].

Both the minimum vertex separation and cutwidth problems are NP-hard on general graphs [Len81, Gav97]. The minimum vertex separation problem remains NP-hard on planar graphs with maximum degree three [MS89, MS88]. The minimum cutwidth problem is also NP-hard on planar graphs with maximum degree three [MS88] and grid graphs [DPPS01].

Both problems have polynomial-time algorithms for trees. The vertex separation of a tree with n vertices can be computed in $O(n)$ time [EST94, Sko00]. The algorithm due to Skodinis [Sko00] also computes a corresponding layout in linear time. The algorithm due to Ellis, Sudborough, and Turner [EST94] takes $O(n \lg n)$ time to compute an optimal layout, where $\lg n$ is the base-2 logarithm of n. The first and only polynomial-time algorithm to date that computes the cutwidth and a corresponding optimal layout of a tree runs in $O(n \lg n)$ time [Yan85]. The minimum vertex separation problem is also solvable in polynomial time on graphs of bounded treewidth [BK96] and multidimensional grids [BL91]. The classes of graphs for which the cutwidth problem has a polynomial-time algorithm include hypercubes [Har64] and graphs of bounded treewidth and degree [TSB01].

The main results in this thesis are four algorithms that update the vertex separation of a rooted tree after a subtree is either attached to or removed from it. In order to perform such an update quickly, certain additional information is stored and updated at all vertices of the tree. The first algorithm solves the special case when two trees are joined at their roots. It runs in $O(\lg n)$ time, where n is the number of vertices in the resulting tree. The algorithm is described in Chapter 5. The second algorithm (Chapter 6) updates the vertex separation of a tree after a subtree of the root is removed; it runs in $O\left(\lg^2 n\right)$ time and requires an approach quite different from that used in the first algorithm. We remark that in this case, n equals the number of vertices in the original tree, not in the tree resulting from the subtree removal.

The third and fourth algorithms, presented in Chapter 7, solve the more general problems of updating the vertex separation after a tree is attached to an arbitrary vertex of another tree, and after a subtree of an arbitrary vertex is removed from the tree, respectively. The algorithms run in $O\left(m^{\log_3 2} \lg^2 m\right)$ amortized time over a sequence of m tree additions or subtree removals, provided that each tree added or subtree removed has a constant size independent of m. We emphasize that all these bounds are strictly sublinear, making our algorithms asymptotically faster than the current algorithms for computing the vertex separation of a tree [EST94, Sko00].

This thesis also presents and proves lower and upper bounds on the vertex separation and cutwidth of a general graph (Chapter 3). Both upper bounds are given in terms of the number of vertices in the graph; the lower bound on vertex separation is given in terms of the minimum degree of a vertex in the graph, whereas the lower bound on cutwidth is a function of the maximum vertex degree in the graph.

Before presenting our results, we review basic notation, terminology, and results that are used extensively in the rest of the thesis.

Chapter 2

Definitions, Notation, and Basic Results

2.1 Layouts, Vertex Separation, and Cutwidth

A *linear layout* φ, or simply *layout*, *of a graph* $G = (V_G, E_G)$ is a bijection from V_G to the set of integers $\{1, \ldots, n = |V_G|\}$. An integer i in the range of φ is also called the *ith position of* φ, or simply a *position of* φ. Since layout φ is a bijection, the inverse function φ^{-1} is well-defined. The vertices $\varphi^{-1}(1)$ and $\varphi^{-1}(n)$ are called the *leftmost* and *rightmost vertices of layout* φ, respectively.

The goal of a *layout problem on graph* G is to find a layout of G so that a certain objective function defined on graph G and layout φ is minimized. We will define two such objective functions shortly. We remark that there are $n!$ possible layouts of graph G, so a brute-force search strategy over all layouts to minimize an objective function is not practical, except for very small values of n.

Díaz, Petit, and Serna [DPS02] give a nice summary of graph layout problems in a coherent framework. We will only need a small subset of their notation and definitions, which we discuss below. We can associate various quantities with graph G and layout φ that measure the "goodness" of the layout. Before introducing two such quantities (vertex separation and cutwidth), it is convenient to define a few auxiliary variables.

Many objective functions measuring how "good" a layout φ is are maxima over all integers i, $1 \leq i \leq n$, of a quantity that relates vertices u to vertices v where $\varphi(u) \leq i$ and $\varphi(v) > i$. We therefore define the *left side of position* i, denoted by $L(i, \varphi, G)$, as the set of all vertices to the left of and including the vertex at position i of φ; that is, $L(i, \varphi, G) = \{u \in V_G : \varphi(u) \leq i\}$. Similarly, the *right side of position* i, denoted by $R(i, \varphi, G)$, is the set of those vertices that lie to the right of the vertex at the ith position of layout φ; formally, $R(i, \varphi, G) = \{v \in V_G : \varphi(v) > i\}$. We remark that this excludes the vertex at that position, in contrast to the definition of $L(i, \varphi, G)$; we denote by $\widetilde{L}(i, \varphi, G)$ the symmetric analogue of $R(i, \varphi, G)$, that is, $\widetilde{L}(i, \varphi, G) = L(i, \varphi, G) - \{\varphi^{-1}(i)\}$. Sets $L(i, \varphi, G)$ and $R(i, \varphi, G)$ form a binary partition of vertex set V_G. Figure 2.1 shows a graph G and one of its layouts φ with $\varphi(v_5) = 1$, $\varphi(v_6) = 2$, $\varphi(v_1) = 3$, and so on; also, $L(4, \varphi, G) = \{v_5, v_6, v_1, v_2\}$ and $R(4, \varphi, G) = \{v_3, v_4, v_7\}$.

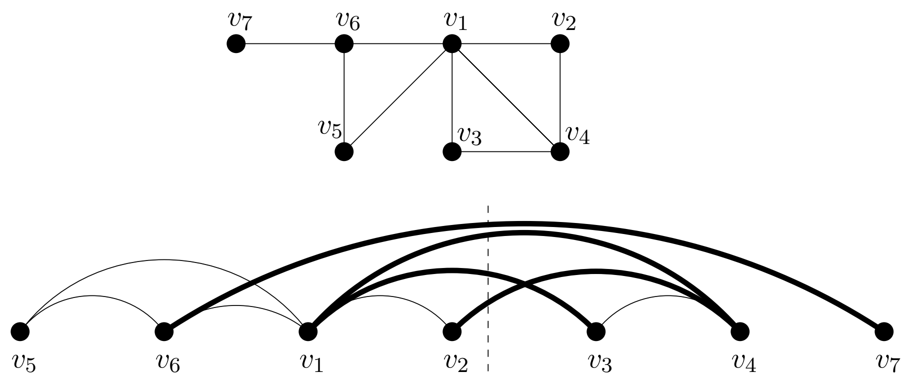

FIGURE 2.1: A graph G on seven vertices and one of its layouts φ. The dashed line separates sets $L(4, \varphi, G)$ and $R(4, \varphi, G)$.

In general, a measure that relates the vertices in set $L(i, \varphi, G)$ to those in set $R(i, \varphi, G)$ has low value if $L(i, \varphi, G)$ and $R(i, \varphi, G)$ are well-separated in some sense. Two such measures are quite natural: the number of edges with endpoints in different sets, and the number of vertices in one set that are adjacent to vertices in the other set. An edge is said to *cross over position* i *in layout* φ if one of its endpoints belongs to $L(i, \varphi, G)$ and the other endpoint belongs to $R(i, \varphi, G)$. In Figure 2.1, all edges that cross over position 4 are drawn with thick lines. The *edge cut at position* i, denoted $\theta(i, \varphi, G)$, is the number of edges that cross over position

i; that is, $\theta(i, \varphi, G) = |\{uv \in E_G : u \in L(i, \varphi, G) \text{ and } v \in R(i, \varphi, G)\}|$. The layout in Figure 2.1 has $\theta(4, \varphi, G) = 4$. The *cutwidth of layout* φ is the maximum edge cut over all positions of φ; in other words, $\mathrm{CW}(\varphi, G) = \max_{1 \leq i \leq n} \theta(i, \varphi, G)$. In the figure, the maximum edge cut of four occurs at positions 3, 4, and 5 of φ (vertices v_1, v_2, and v_3), and therefore $\mathrm{CW}(\varphi, G) = 4$.

The objective of the *minimum cutwidth problem* is to find a layout φ^* with the minimum value of $\mathrm{CW}(\varphi^*, G)$; that is, we want a layout satisfying $\mathrm{CW}(\varphi^*, G) = \min_\varphi \mathrm{CW}(\varphi, G)$. This minimum value is called the *cutwidth of graph* G and is denoted by $\mathrm{CW}(G)$. A layout of G whose cutwidth is $\mathrm{CW}(G)$ is said to be *optimal with respect to cutwidth*, or simply *optimal* if the minimization requirement with respect to cutwidth is clear from context.

The next quantity that we define measures the separation of sets $L(i, \varphi, G)$ and $R(i, \varphi, G)$ in terms of vertex adjacency. The *vertex cut at position* i, denoted $\delta(i, \varphi, G)$, equals the number of vertices in $L(i, \varphi, G)$ that are adjacent to vertices in $R(i, \varphi, G)$. In other words, the vertex cut at position i is the number of vertices in set $L(i, \varphi, G)$ that are incident to edges that cross over position i. We can formally define this measure as $\delta(i, \varphi, G) = |\{u \in L(i, \varphi, G) : (\exists v \in R(i, \varphi, G) : uv \in E_G)\}|$. In Figure 2.1, the vertices in the left side of position 4 that satisfy this condition are v_6, v_1, and v_2. Therefore, $\delta(4, \varphi, G) = 3$. The *vertex separation of layout* φ is defined as the maximum vertex cut over all positions: $\mathrm{VS}(\varphi, G) = \max_{1 \leq i \leq n} \delta(i, \varphi, G)$. In the layout of Figure 2.1, the maximum vertex cut occurs at position 5 (vertex v_3) and is equal to four: $\mathrm{VS}(\varphi, G) = 4$.

In the *minimum vertex separation problem*, we want to find a layout φ^* that minimizes the function $\mathrm{VS}(\varphi^*, G)$; that is, $\mathrm{VS}(\varphi^*, G) = \min_\varphi \mathrm{VS}(\varphi, G)$. We call the vertex separation of φ^* the *vertex separation of graph* G, and denote it by $\mathrm{VS}(G)$. Layout φ^* is *optimal with respect to vertex separation*, or simply *optimal* if the context of vertex separation is obvious. We also say that a *layout corresponds to a particular value of vertex separation* (*cutwidth*) if it achieves that value of vertex separation (cutwidth).

We observe that the vertex cut at a position i of layout φ is bounded above by the edge cut at i, since for each vertex in set $L(i, \varphi, G)$ that is adjacent to a vertex in set $R(i, \varphi, G)$, there is an edge crossing over position i. We conclude that

$\mathrm{VS}(G) \leq \mathrm{CW}(G)$. It is not obvious what an upper bound on $\mathrm{CW}(G)$ in terms of $\mathrm{VS}(G)$ looks like, and it does not seem to have been investigated. It is certainly an interesting question. We remark it is possible that the vertex cut at position 1 of layout φ is 1, while the edge cut at this position is $n-1$; this situation occurs when vertex $\varphi^{-1}(1)$ is adjacent to all other $n-1$ vertices in graph G.

The layout in Figure 2.1 is optimal with respect to neither vertex separation nor cutwidth. To see this, consider placing vertex v_7 between vertices v_6 and v_1 in the layout, as shown in Figure 2.2. Both the vertex separation and cutwidth of this new layout are 3, as opposed to 4 in the old layout.

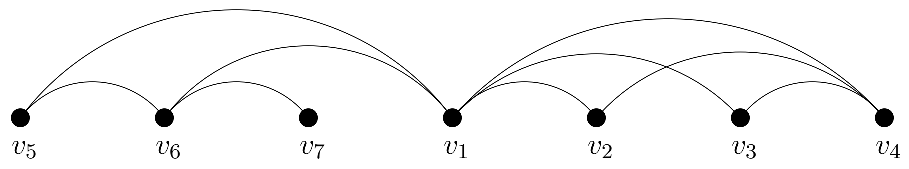

FIGURE 2.2: An alternative layout of the graph in Figure 2.1 with smaller values of vertex separation and cutwidth.

Optimal layouts with respect to vertex separation and cutwidth are in general different. Also, an optimal layout need not be unique: several different layouts of a graph can have the same minimum vertex separation or cutwidth. An extreme example of this fact is a graph with no edges, every layout of which is optimal with both vertex separation and cutwidth 0.

We now state and prove an important lemma that appears in the survey of Díaz, Petit, and Serna [DPS02]. The lemma below relates the vertex separation and cutwidth of a graph and its subgraph. It is the foundation of most results that come afterward.

Lemma 2.1 If H is a subgraph of a graph G, then $\mathrm{VS}(H) \leq \mathrm{VS}(G)$ and $\mathrm{CW}(H) \leq \mathrm{CW}(G)$. □

We next review some terminology and prove a few simple results for graphs that will be needed later in this thesis.

2.2 Graphs

For a graph $G = (V_G, E_G)$, the minimum degree of any vertex in G is denoted by $\varepsilon(G)$. Likewise, $\Delta(G)$ denotes the maximum degree of a vertex in graph G. The number of vertices in G is denoted by $|G|$. The *subgraph H of graph G induced by set $V_H \subseteq V_G$* is the graph with vertex set V_H and edge set E_H, where an edge uv belongs to E_H if and only if $uv \in E_G$ and $\{u, v\} \subseteq V_H$. The *trivial graph*, or *trivial tree*, is the graph having only one vertex and no edges. The *empty graph*, or *empty tree*, is the graph with no vertices. The vertex separation and cutwidth of the empty graph are defined to be 0. A *nonempty graph* is a graph that is not empty.

Lemma 2.2 If G is a connected graph, then $\mathrm{VS}(G) = 0$ and $\mathrm{CW}(G) = 0$ if and only if G is the trivial or empty graph.

Proof. We prove the lemma by assuming the left side of the equivalence first, and then assuming the right side. First, assume $\mathrm{VS}(G) = 0$ and $\mathrm{CW}(G) = 0$, and suppose that G contains edge uv, so that it is not the empty or trivial graph. Given an optimal layout φ of G with respect to vertex separation (cutwidth), we may assume without loss of generality that $\varphi(u) < \varphi(v)$. Then the vertex cut (edge cut) at position $\varphi(u)$ of φ is at least 1, which is a contradiction. Hence, G has no edges and therefore is the trivial or empty graph; if G had more than one vertex, it would not be connected. Second, assume graph G is the trivial or empty graph. If G is the empty graph, then its vertex separation and cutwidth are both 0 by definition. If G is the trivial graph containing the only vertex w, then there is only one layout φ of G: $\varphi(w) = 1$. Both the vertex cut and edge cut at position 1 of φ are 0, and thus we conclude that $\mathrm{VS}(G) = 0$ and $\mathrm{CW}(G) = 0$ hold. $\square$

Our proofs of many results on the vertex separation of a tree require reasoning about paths in the tree. A *path in a graph G* is a sequence $P = u_1, u_2, \ldots, u_m$ of vertices in G such that there is an edge in G between each pair of consecutive vertices in P. Path P is a *simple path* if $u_1, \ldots, u_m$ are all distinct vertices. We are concerned exclusively with simple paths in this thesis. Path P is *oriented* in the sense that u_1 is the first vertex and u_m is the last vertex of P. In many cases, it does not matter what the orientation of a path is, and so P and $P^R = u_m, u_{m-1}, \ldots, u_1$ are the same path. However, in order to reason about a path, we need to fix its orientation; we often fix the orientation of the path in a way that is most convenient.

We next introduce terminology to refer to specific vertices in path P. The *endpoints of* P are vertices u_1 and u_m; u_1 is the *left endpoint* and u_m is the *right endpoint.* They are denoted by first_P and last_P, respectively. We note that if $m = 1$, then $\mathit{first}_P = \mathit{last}_P$, and hence path P has only one endpoint. Vertex u_i is called an *interior vertex of path* P if $2 \leq i \leq m - 1$. An interior vertex u_i of P has the *left neighbour* u_{i-1}, denoted $\mathit{left}(u_i, P)$, and the *right neighbour* u_{i+1}, denoted $\mathit{right}(u_i, P)$. If $m > 1$, the *neighbours of endpoints* u_1 and u_m are the vertices u_2 and u_{m-1}, respectively. The only neighbour of an endpoint u_i of P is denoted by $\mathit{right}(u_i, P)$ or $\mathit{left}(u_i, P)$, depending on whether $i = 1$ or $i = m$, respectively.

We next extend the notions of adjacency and incidence to paths. Vertex u_i is said to *be in*, or *on*, *path* P for all i, $1 \leq i \leq m$. An edge is said to *be in* P if its endpoints are consecutive vertices in P. An *edge is incident to path* P if one of its endpoints is in P, but the other endpoint is not. A vertex is *adjacent to path* P if it is not in P and is an endpoint of an edge incident to P. The *length of path* P, denoted $|P|$, is equal to the number of edges in P; in other words, it is equal to the number of vertices in P minus one. We occasionally say that a *vertex* u *is closer to a vertex* v *than it is to a vertex* w; this means that the length of a shortest path from u to v is smaller than the length of a shortest path from u to w in graph G.

We occasionally need to refer to the subgraph of graph G induced by the vertices in path P. The *path graph on* m *vertices*, denoted P_m, consists of vertices $u_1, \ldots, u_m$ and edges $u_i u_{i+1}$ for all i such that $1 \leq i \leq m - 1$. If we specify P_m by a sequence of vertices, instead of by vertex and edge sets, then a path subgraph of graph G is a simple path in G. Because of this correspondence, we use the concepts of path subgraph and simple path interchangeably.

Lemma 2.3 The vertex separation and cutwidth of the path graph P_n are 1 for all $n \geq 2$.

Proof. We describe a layout φ of P_n with vertex separation and cutwidth 1 if $n \geq 2$. Then, since P_n is connected and is neither the empty nor trivial graph by the fact that $n \geq 2$, Lemma 2.2 implies $\mathrm{VS}(P_n) > 0$ and $\mathrm{CW}(P_n) > 0$. We can thus conclude that the vertex separation and cutwidth of P_n are 1 for all $n \geq 2$, because $\mathrm{VS}(P_n) \leq \mathrm{VS}(\varphi, P_n) = 1$ and $\mathrm{CW}(P_n) \leq \mathrm{CW}(\varphi, P_n) = 1$.

It remains to show that there is a layout φ of graph P_n with vertex separation and cutwidth 1. Writing $P_n = u_1, \ldots, u_n$, we define φ by $\varphi(u_i) = i$ for all i, $1 \leq i \leq n$. Both the vertex cut and edge cut at position n are 0, since $R(n, \varphi, P_n) = \emptyset$. Therefore, we only consider positions i such that $1 \leq i \leq n-1$, and show that both the vertex cut and edge cut at these positions are 1. Vertex u_i is incident to exactly two edges $u_{i-1}u_i$ and u_iu_{i+1} if $2 \leq i \leq n-1$, and to exactly one edge u_iu_{i+1} if $i = 1$. Furthermore, there is no vertex u_j in P_n such that $j < i - 1$ that is adjacent to a vertex in $R(i, \varphi, P_n) = \{u_{i+1}, \ldots, u_n\}$. Hence, the only edge that crosses over position i is the edge u_iu_{i+1}. We conclude that both the vertex cut and edge cut at position i of layout φ are 1, and therefore $\mathrm{VS}(\varphi, P_n) = \mathrm{CW}(\varphi, P_n) = 1$. □

We now introduce notation for combining paths together to form longer paths. A *subpath of path P* is a contiguous subsequence of P. Path Q is a *proper subpath of* P if it is a subpath of P and $Q \neq P$. When the orientation of path P is unimportant, then we may say that a subpath Q of path P is also a subpath of path P^R. Given path $Q = v_1, \ldots, v_n$ in graph G such that u_mv_1 is an edge in G, we denote by $P + Q$ the path $u_1, \ldots, u_m, v_1, \ldots, v_n$. Given a vertex w in G such that wu_1 is an edge, path $w + P$ is $w, u_1, \ldots, u_m$. Similarly, if wu_m is an edge in G, path $P + w$ is $u_1, \ldots, u_m, w$. Finally, if u_i is an interior vertex of path P, we sometimes write $P = \mathit{lpath}(u_i, P) + u_i + \mathit{rpath}(u_i, P)$, where $\mathit{lpath}(u_i, P) = u_1, \ldots, u_{i-1}$ and $\mathit{rpath}(u_i, P) = u_{i+1}, \ldots, u_m$ are the *subpaths of P to the left* and *right of vertex u_i*, respectively. In order to make the path algebra easier, we define the *empty path* to be the path $\mathcal{O}$ satisfying $P = P + \mathcal{O}$ and $P = \mathcal{O} + \mathcal{P}$. A path is *nonempty* if it is not empty.

In proving lower and upper bounds on vertex separation and cutwidth in Chapter 3, we will need two special classes of graphs. The *complete graph on n vertices*, denoted K_n, is the graph containing n vertices and having an edge between every pair of vertices. The *star graph on $n + 1$ vertices*, referred to as S_n, contains one vertex of degree n to which all other n vertices are adjacent, and there is no edge between any two of these vertices. We emphasize that graph K_n contains n vertices, whereas graph S_n contains $n + 1$ vertices.

We have covered all the notation and terminology regarding general graphs that will be needed. We now focus on trees, which play a central role in this thesis.

2.3 Trees

The tree is a prevalent structure in computer science, and it is often the first nontrivial class of graphs on which a polynomial-time algorithm for a new graph problem is found. Many of the definitions applying to trees that will be used later are defined in this section. We also state a few simple results about the properties of trees. We first focus on unrooted trees, and then move on to rooted trees. Although the vertex separation of a tree is defined on an unrooted tree, the algorithms to compute it require the tree to be rooted at an arbitrary vertex.

A *tree* T is a graph such that for any two vertices u and v in T, there exists a unique simple path P with left endpoint u and right endpoint v. Since the path P is unique, it is denoted by $sp(u, v)$. This definition of 'tree' is equivalent to the graph being acyclic [CLR90]; a *graph* G *is acyclic* if there is no path P in G of length at least 3 such that the endpoints of P are identical, and all other vertices in P are distinct. A collection of trees forms a graph, which is not necessarily connected, called a *forest*.

In relating the vertex separation of tree T to its structure, we will consider a path P in T whose removal from T produces a forest, each tree of which has vertex separation less than $\mathrm{VS}(T)$. It is therefore convenient to introduce notation and terminology to refer to the trees in this forest. Given tree T and one of its vertices u, the *branches of* u *in* T are the connected components of the forest obtained by removing u and its incident edges from tree T. The set of all branches of vertex u in T is denoted by $T[u]$. This definition naturally extends to a path in the tree: the set of *branches of path* P *in tree* T consists of the connected components of the forest obtained by removing from T all vertices in P and all edges in and incident to P; it is denoted by $T[P]$. Figure 2.3 shows tree T and the branches T_1, T_2, T_3, and T_4 of path $P = u, v, w$ in T. If vertices u and z are adjacent in T, then the branch of u containing z is denoted by $T[u]_z$. In the figure, $T[v]_z = T_3$. Similarly, if path P and vertex z are adjacent, then $T[P]_z$ is the branch of P containing z.

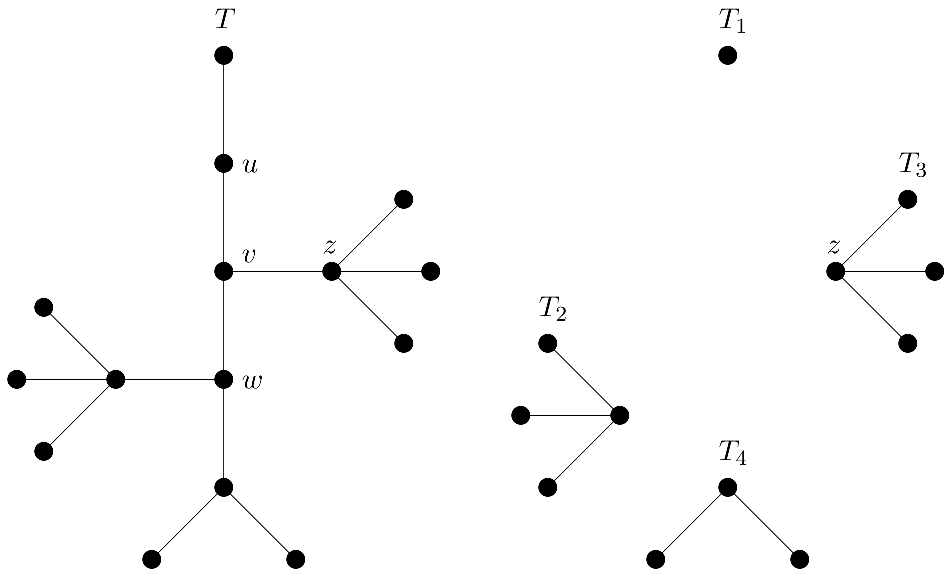

FIGURE 2.3: Tree T and the branches T_1, T_2, T_3, and T_4 of path $P = u, v, w$.

Thus, $T[P]_z = T_3$ in the figure.

Having discussed how to select a particular branch out of the set of all branches of a vertex or path in a tree, we now discuss subsets of the set of branches. If u is a vertex on path P in tree T, then $T[P] \cap T[u]$ is the set of branches R in set $T[P]$ with a vertex in R adjacent to vertex u. In Figure 2.3, $T[P] \cap T[u] = \{T_1\}$, $T[P] \cap T[v] = \{T_3\}$, and $T[P] \cap T[w] = \{T_2, T_4\}$. We also observe that for a subpath Q of path P, set $T[P] \cap T[Q]$ is the subset of $T[P]$ of all branches R such that there is a vertex in R adjacent to path Q. In the figure, $T[P] \cap T[u, v] = \{T_1, T_3\}$ and $T[P] \cap T[v, w] = \{T_3, T_2, T_4\}$.

In proving in Chapter 4 the uniqueness of the shortest path P in tree T such that all branches of P have vertex separation less than $\mathrm{VS}(T)$, we will need to relate the set of branches of P to the set of branches of a subpath of P. We therefore state three lemmas that describe this relationship. In the first two lemmas, we do this by relating the set of branches of a vertex u in path P to the set of branches of P. In the first lemma, we handle the case when u is an endpoint of path P. Since we apply this result in cases when the orientation of P is unimportant, we only consider the case when u is the left endpoint of path P.

Lemma 2.4 Consider a simple path $P = u_1, \ldots, u_m$ in a tree T such that $m > 1$. The following equalities hold:

1. $T[u_1] = (T[P] \cap T[u_1]) \cup \{T[u_1]_{u_2}\}$ and
2. $T[u_1] = (T[P] - T[\mathit{rpath}(u_1, P)]) \cup \{T[u_1]_{u_2}\}$.

Proof. Figure 2.4 illustrates the essential elements of the proof. The first equation

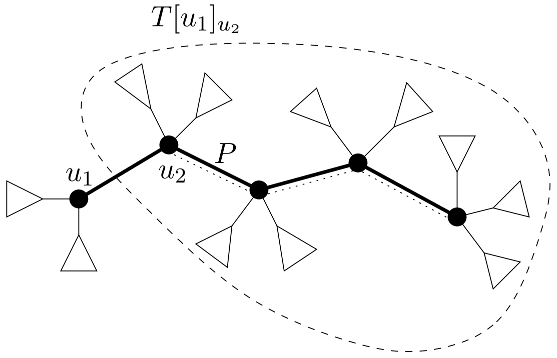

FIGURE 2.4: The proof of Lemma 2.4. The subpath $\mathit{rpath}(u_1, P)$ of path P is indicated with a dotted line.

is a consequence of the fact that each branch of vertex u_1 in tree T is either a branch of path P that contains a vertex adjacent to u_1 (that is, it is a branch in set $T[P] \cap T[u_1]$), or it is the branch $T[u_1]_{u_2}$ containing the subpath $\mathit{rpath}(u_1, P)$ of P. The second equation is derived as follows. Every branch of path P is also a branch of vertex u_1, except when it is also a branch of subpath $\mathit{rpath}(u_1, P)$ of P; the excluded branch is therefore in set $T[\mathit{rpath}(u_1, P)]$. Furthermore, every branch of vertex u_1 is also a branch of path P, except branch $T[u_1]_{u_2}$, which contains path $\mathit{rpath}(u_1, P)$. □

In the next lemma, we consider the case when u is an interior vertex of path P.

Lemma 2.5 Consider a simple path $P = u_1, \ldots, u_m$ in a tree T such that $m \geq 3$ and an interior vertex u_i of P. The following equalities hold:

1. $T[u_i] = (T[P] \cap T[u_i]) \cup \left\{T[u_i]_{u_{i-1}}, T[u_i]_{u_{i+1}}\right\}$ and
2. $T[u_i + \mathit{rpath}(u_i, P)] = (T[P] - T[\mathit{lpath}(u_i, P)]) \cup \left\{T[u_i]_{u_{i-1}}\right\}$.

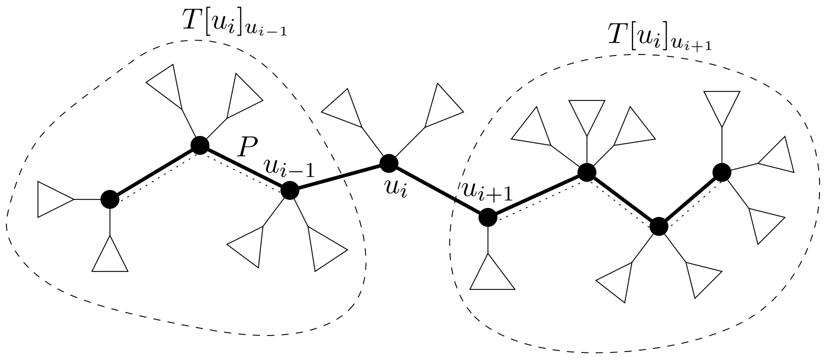

FIGURE 2.5: The proof of Lemma 2.5. The subpaths $\mathit{lpath}(u_i, P)$ and $\mathit{rpath}(u_i, P)$ of path P are indicated with dotted lines.

Proof. Figure 2.5 illustrates the proof. The first equation follows from observing that a branch of vertex u_i in tree T is either a branch of path P that contains a vertex adjacent to u_i, or it is the branch $T[u_i]_{u_{i-1}}$ or branch $T[u_i]_{u_{i+1}}$ containing the subpath $\mathit{lpath}(u_i, P)$ or $\mathit{rpath}(u_i, P)$ of P, respectively. The second equation is a consequence of the following facts. Every branch in set $T[P]$ is also a branch of subpath $u_i + \mathit{rpath}(u_i, P)$, except when it is also a branch of subpath $\mathit{lpath}(u_i, P)$; the excluded branch is therefore in set $T[\mathit{lpath}(u_i, P)]$. In addition, each branch of path $u_i + \mathit{rpath}(u_i, P)$ is also a branch of path P, except when it is the branch $T[u_i]_{u_{i-1}}$, which contains the subpath $\mathit{lpath}(u_i, P)$ of P. □

Finally, we investigate the relationship between the branches of a path and the branches of its subpaths.

Lemma 2.6 Consider a simple path $P = u_1, \ldots, u_m$ in a tree T such that $m > 1$, and subpaths $P_l = u_1, \ldots, u_i$ and $P_r = u_{i+1}, \ldots, u_m$ of P, where $1 \leq i \leq m - 1$; that is, paths P_l and P_r are nonempty. The following three statements hold:

1. all branches of P_l in T except $T[P_l]_{u_{i+1}}$ are subtrees of the branch $T[P_r]_{u_i}$,

2. all branches of P_r in T except $T[P_r]_{u_i}$ are subtrees of the branch $T[P_l]_{u_{i+1}}$, and

3. $T[P] = (T[P_l] \cup T[P_r]) - \left\{T[P_l]_{u_{i+1}}, T[P_r]_{u_i}\right\}$.

Proof. Figure 2.6 illustrates the proof. All branches of subpath P_l in tree T are subtrees of branch $T[P_r]_{u_i}$, except branch $T[P_l]_{u_{i+1}}$, which contains subpath P_r. This proves the first statement. Similarly, all branches of subpath P_r in T are

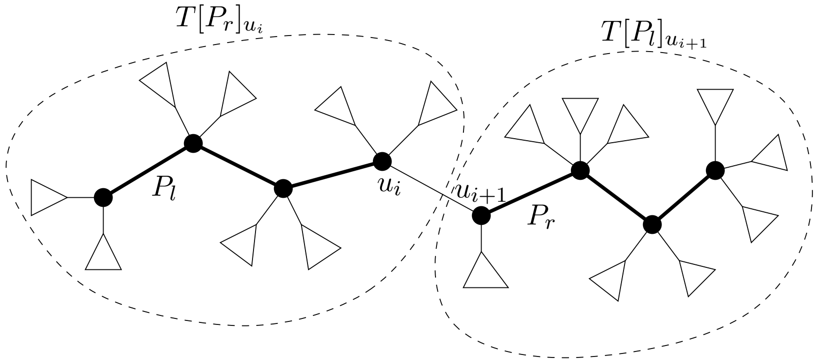

FIGURE 2.6: The proof of Lemma 2.6.

subtrees of branch $T[P_l]_{u_{i+1}}$, except branch $T[P_r]_{u_i}$, which contains P_l. This proves the second statement. We also see from the figure that every branch of path P is a branch of either subpath P_l or subpath P_r, and that every branch of either P_l or P_r is a branch of P, except the branches $T[P_l]_{u_{i+1}}$ and $T[P_r]_{u_i}$ containing P_r and P_l, respectively. This proves the third statement. □

As mentioned earlier, the algorithms that compute the vertex separation of a tree take as input rooted trees. We have so far discussed concepts that are independent of whether the tree is rooted or not. In the remainder of this chapter, we discuss concepts related to rooted trees. A tree T is *rooted at* u if a vertex u in T is designated as the *root of* T. The root of T is denoted by r_T. A tree that is not rooted is called *unrooted*. Rooting a tree corresponds to imposing a partial order, called the *ancestor-descendant relationship*, on the vertices in T. A vertex v in tree T is an *ancestor of a vertex* w *in* T if path $sp(r_T, v)$ is a subpath of path $sp(r_T, w)$. Vertex w is then called a *descendant of vertex* v. If $|sp(r_T, v)| < |sp(r_T, w)|$, then v is a *proper ancestor of* w, and w is a *proper descendant of* v.

If vertex v is an ancestor of vertex w in tree T and vw is an edge in T, then w is a *child of* v *in* T, and v is the *parent of* w *in* T; the parent is denoted by p_w. The *rooted degree of vertex* v *in* T is the number of children of v. The *unrooted degree of*

v, or simply the *degree of* v, is the number of vertices in tree T adjacent to vertex v. Vertex v is called a *leaf* if its rooted degree is 0. Otherwise, v is an *internal vertex of* T. Tree T is called (*partially*) *ordered* if there is a (partial) order defined on the children of each internal vertex; otherwise, T is *unordered.* The *depth of a vertex* v *in tree* T is defined to be $|sp(r_T, v)|$. The *height of tree* T is the maximum depth of a vertex in T. Rooted tree T is called a *perfect tree* if all internal vertices of T have the same rooted degree and the depth of all leaves of T is the same.

Subtrees of a rooted tree T naturally inherit the ancestor-descendant relationship of T. The *subtree of* T *rooted at vertex* u is the tree induced by the descendants of u in T and having root u; it is denoted by T_u. Tree T_u is also called a *rooted subtree of* T. We emphasize that while every rooted subtree of rooted tree T is a subtree of T, not every subtree of T is a rooted subtree of T; a rooted subtree is special in that it contains all descendants in T of its root. We note that $T = T_{r_T}$. The *subtrees yielded by a vertex* u *in tree* T are the subtrees rooted at the children of u. Figure 2.7 shows the subtree T_u of tree T, and subtrees T_1, T_2, and T_3 yielded by vertex u. When we say that a *rooted tree* T *is a subtree of another rooted tree*

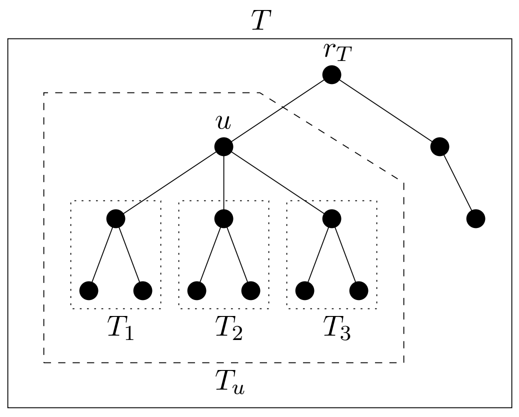

FIGURE 2.7: Subtree T_u of tree T, and the subtrees T_1, T_2, and T_3 yielded by vertex u.

T', then not only is the unrooted tree corresponding to T required to be a subtree of the unrooted tree corresponding to T', but also all the descendants of the root of T in T' must be in T.

Although vertex separation and cutwidth are defined on unrooted trees, the algorithms that compute these measures work on rooted trees [EST94, Sko00, Yan85]. It does not matter which vertex is the root. The algorithms start execution at the root and apply recursion on its children. When we talk about a rooted tree, it is assumed that the tree has been rooted at an arbitrary vertex. Rooting a tree does not change its structure, and therefore the vertex separation and cutwidth of a rooted tree and the corresponding unrooted tree are the same.

We next introduce notation for representing a rooted tree from which several rooted subtrees have been removed. The notation is a minor modification of the notation used by Ellis, Sudborough, and Turner in their work on the vertex separation of trees [EST94]. Given rooted tree T, we denote by $T\langle u_1\rangle$ the tree obtained from T by removing rooted subtree T_{u_1}; this assumes vertex u_1 is in tree T. We can continue removing subtrees from T; the tree $T\langle u_1\rangle\langle u_2\rangle$ is obtained by removing subtree S_{u_2} from tree $S = T\langle u_1\rangle$, again assuming vertex u_2 is in S. To reduce clutter, we contract $T\langle u_1\rangle\langle u_2\rangle$ to $T\langle u_1, u_2\rangle$. Figure 2.8 illustrates this notation. We now make the definition precise.

Definition 2.1 [EST94] We denote by $T\langle u_1, \ldots, u_p\rangle$ the tree obtained from a rooted tree T by applying the following recursive procedure:

1. tree $T\langle u_1\rangle$ is obtained from T by removing subtree T_{u_1}, where u_1 is a vertex in T; and

2. for all i, $2 \leq i \leq p$, tree $T\langle u_1, \ldots, u_i\rangle$ is obtained from tree $S = T\langle u_1, \ldots, u_{i-1}\rangle$ by removing subtree S_{u_i}, assuming u_i is a vertex in tree S.

It is useful to combine notation T_u and $T\langle u_1, \ldots, u_p\rangle$. Notation $T_u\langle u_1, \ldots, u_p\rangle$ is parsed as $S\langle u_1, \ldots, u_p\rangle$, where $S = T_u$. Notation $T\langle u_1, \ldots, u_p\rangle_u$ means S_u, where $S = T\langle u_1, \ldots, u_p\rangle$. For notational convenience, we also define $T\langle\rangle$ to be the tree T.

We now make a few observations about tree $T\langle u_1, \ldots, u_p\rangle$ to gain more intuitive understanding of the definition. If u_i is an ancestor of u_j in tree T and $1 \leq j < i \leq p$, then $T\langle u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_p\rangle = T\langle u_1, \ldots, u_p\rangle$. This is because removing subtree $T\langle u_1, \ldots, u_{i-1}\rangle_{u_i}$ removes subtree $T\langle u_1, \ldots, u_{j-1}\rangle_{u_j}$ as well if it has not already been removed, so we do not need to remove $T\langle u_1, \ldots, u_{j-1}\rangle_{u_j}$ explicitly. In Figure 2.8, $T\langle u_2\rangle = T\langle u_1, u_2\rangle$, because u_2 is an ancestor of u_1. Similarly, $T\langle u_2, u_3\rangle =$

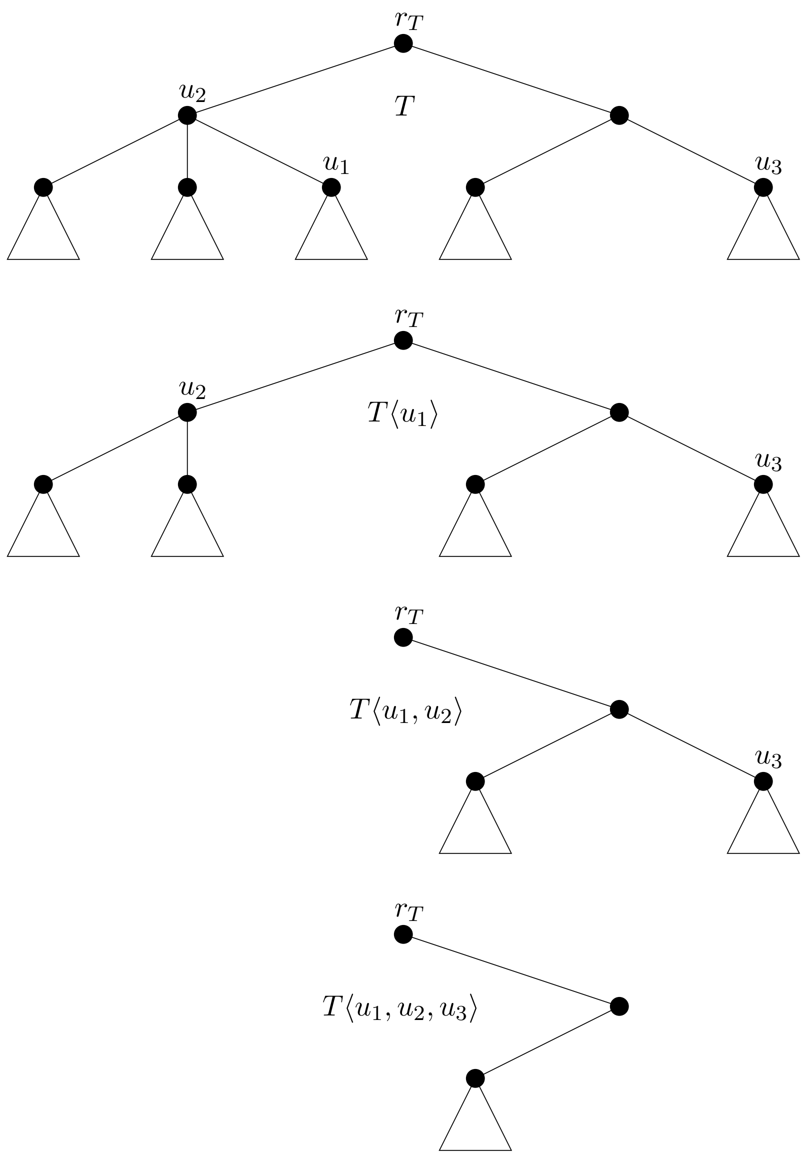

FIGURE 2.8: Trees T, $T\langle u_1\rangle$, $T\langle u_1, u_2\rangle$, and $T\langle u_1, u_2, u_3\rangle$.

$T\langle u_1, u_2, u_3\rangle$. More generally, there is a shortest sequence σ of vertices $v_1, \ldots, v_q$ with $q \leq p$ such that $T\langle v_1, \ldots, v_q\rangle = T\langle u_1, \ldots, u_p\rangle$ and $\{v_1, \ldots, v_q\} \subseteq \{u_1, \ldots, u_p\}$. This sequence is not necessarily unique, as we will see shortly. No vertex in σ is an ancestor of another vertex w in σ; otherwise, we could remove w from sequence σ to form a shorter sequence. This implies for any permutation π of the sequence of integers $1, \ldots, q$, the following equality holds: $T\langle v_1, \ldots, v_q\rangle = T\left\langle v_{\pi(1)}, \ldots, v_{\pi(q)}\right\rangle$. Therefore, sequence σ is not unique if $q > 1$; there are exactly $q!$ such sequences. In Figure 2.8, for example, $T\langle v_1, v_2, v_3\rangle = T\langle v_2, v_3\rangle = T\langle v_3, v_2\rangle$, since v_2 is neither an ancestor nor a descendant of v_3 in tree T. In this case, $\sigma = v_2, v_3$ or $\sigma = v_3, v_2$. In other words, we can treat σ as a set.

Intuitively, removing subtrees rooted at vertices $u_1, \ldots, u_p$ from trees T and S such that T is a subtree of S should preserve the subtree relationship of T and S. This is in fact the case.

Lemma 2.7 If T and S are rooted trees such that T is a subtree of S, and $u_1, \ldots, u_p$ are vertices such that u_1 is in both trees T and S, and for each integer i such that $2 \leq i \leq p$ vertex u_i is in both trees $T\langle u_1, \ldots, u_{i-1}\rangle$ and $S\langle u_1, \ldots, u_{i-1}\rangle$, then tree $T\langle u_1, \ldots, u_p\rangle$ is a subtree of tree $S\langle u_1, \ldots, u_p\rangle$.

Proof. We prove the lemma by induction on the number i of subtrees removed from trees T and S. The base case ($i = 0$) is trivial, since T is a subtree of S by assumption. For the induction step, we assume $0 < i \leq p$ and tree $T\langle u_1, \ldots, u_{i-1}\rangle$ is a subtree of tree $S\langle u_1, \ldots, u_{i-1}\rangle$. We show that tree $T\langle u_1, \ldots, u_i\rangle$ is a subtree of tree $S\langle u_1, \ldots, u_i\rangle$. Vertex u_i is in both $T\langle u_1, \ldots, u_{i-1}\rangle$ and $S\langle u_1, \ldots, u_{i-1}\rangle$ by assumption, and hence trees $T\langle u_1, \ldots, u_i\rangle$ and $S\langle u_1, \ldots, u_i\rangle$ are both defined (point 2 in Definition 2.1). By the induction hypothesis, rooted tree $T\langle u_1, \ldots, u_{i-1}\rangle$ is a subtree of rooted tree $S\langle u_1, \ldots, u_{i-1}\rangle$, and hence $T\langle u_1, \ldots, u_{i-1}\rangle$ with subtree $T\langle u_1, \ldots, u_{i-1}\rangle_{u_i}$ removed is a subtree of $S\langle u_1, \ldots, u_{i-1}\rangle$ with $S\langle u_1, \ldots, u_{i-1}\rangle_{u_i}$ removed. We conclude that tree $T\langle u_1, \ldots, u_i\rangle$ is a subtree of tree $S\langle u_1, \ldots, u_i\rangle$. □

We noted earlier that many of our results require analyzing certain paths in a tree; that is why we introduced fairly extensive notation and terminology for paths. A simple path in a rooted tree is one of two kinds, depending on whether or not one of its interior vertices is an ancestor of all other vertices in the path. This distinction is crucial in understanding how the structure of a rooted tree relates to its vertex separation and to an optimal layout with respect to vertex separation.

Given a simple path $P = u_1, \ldots, u_m$ in rooted tree T, P is called a *monotonic simple path in T* if $u_2, \ldots, u_m$ are all descendants or all ancestors of vertex u_1; in other words, P is a subpath of path $sp(r_T, v)$ for some leaf v in T. Path P is called a *nonmonotonic simple path in tree T* if it is not monotonic, that is, if there exists an integer c such that $2 \leq c \leq m-1$ and $u_1, \ldots, u_{c-1}, u_{c+1}, \ldots, u_n$ are all descendants of u_c in T. The vertex u_i is termed the *inflection vertex of path P in tree T* and is denoted by inf_P. The following two lemmas are simple results and are stated without proof. The first lemma will be needed to show that a monotonic simple path can be extended to the root of tree T, while a nonmonotonic path cannot.

Lemma 2.8 Consider a simple path P in a rooted tree T.

1. If P is nonmonotonic, then inf_P is closer to the root r_T of T than is any other vertex in P.

2. If path P is monotonic, then one of the endpoints of P is closer to r_T than is any other vertex in P. □

The second lemma relates the monotonicity of a path and its subpath.

Lemma 2.9 If a subpath Q of a simple path P in a tree T is nonmonotonic, then path P is nonmonotonic. Furthermore, the inflection vertices of Q and P are identical. □

The algorithms in Chapters 5 and 7 update the vertex separation of a tree constructed from two smaller trees, assuming certain information has been computed for the smaller trees. In discussing the algorithms, it is useful to have special notation representing the composite tree. Given two rooted trees T and S with roots r_T and r_S, respectively, $T \vdash S$ is the unordered rooted tree obtained by making r_S a child of r_T. The root of tree $T \vdash S$ is r_T. More generally, if u is any vertex in tree T, then $T \vdash_u S$ is the unordered rooted tree formed by making vertex r_S a child of u. It follows that $T \vdash S = T \vdash_{r_T} S$. If T is the empty tree, then $T \vdash S$ is defined to be the empty tree as well, regardless of whether tree S is empty or not. We observe that if u is a child of root r_S, then $T = T\langle u \rangle \vdash T_u$.

In this chapter, we discussed basic notions that will be needed in the rest of this thesis. We first introduced two measures of a graph, vertex separation and cutwidth, associated with linear layouts of the graph. We then showed that both measures are monotonic under the subgraph relation; that is, the vertex separation and cutwidth of a graph are at least as large as the vertex separation and cutwidth, respectively, of any subgraph.

Since paths play an important role in our work, we developed extensive notation and terminology for them. We stated three lemmas that relate the branches of a path to the branches of its subpath. Subsequently, we introduced rooted trees and reviewed special notation to represent a rooted tree from which several rooted subtrees have been removed. Finally, we introduced monotonic and nonmonotonic paths in a rooted tree, and developed notation for representing a tree constructed from two smaller trees.

Chapter 3

Bounds on Vertex Separation and Cutwidth

In this chapter, we state and prove two theorems on upper and lower bounds on the vertex separation and cutwidth of general graphs. The first theorem gives lower and upper bounds on the vertex separation of a graph, while the second theorem gives analogous bounds on the cutwidth of a graph. To the best of our knowledge, these bounds have not been derived before.

We first investigate the relationship between the vertex separation of a graph G and the number of vertices and minimum degree of a vertex in G. Toward this end, we derive the value for the vertex separation of the complete graph.

Lemma 3.1 The vertex separation of the complete graph K_n equals $n-1$ for any $n \geq 1$.

Proof. We only need to observe that in any layout φ with respect to vertex separation of any graph on n vertices, there can be at most $n-1$ vertices that are adjacent to a vertex to the right of them. Therefore, $\mathrm{VS}(K_n) \leq n-1$ holds. Furthermore, since in complete graph K_n every vertex is adjacent to every other vertex, it follows that $\delta(n-1, \varphi, K_n) = n-1$. We conclude that the vertex separation of complete graph K_n is $n-1$. $\square$

Having shown a simple lemma regarding the vertex separation of graph K_n, we now use this result to derive bounds on the vertex separation of a general graph.

Theorem 3.2 Any nonempty graph G satisfies $\varepsilon(G) \leq \mathrm{VS}(G) \leq |G| - 1$, where $\varepsilon(G)$ is the minimum degree of a vertex in G. Furthermore, these inequalities are tight.

Proof. We first prove that $\mathrm{VS}(G) \leq |G| - 1$. Since every graph is a subgraph of complete graph $K_{|G|}$, it follows from Lemmas 2.1 and 3.1 that $\mathrm{VS}(G) \leq |G| - 1$. Furthermore, Lemma 3.1 implies this inequality is tight.

Next, we show that $\varepsilon(G) \leq \mathrm{VS}(G)$ by considering the last vertex $\varphi^{-1}(|G|)$ of an optimal layout φ of graph G with respect to vertex separation. Vertex $\varphi^{-1}(|G|)$ is adjacent to at least $\varepsilon(G)$ vertices to the left of it, since every vertex in G is adjacent to at least $\varepsilon(G)$ vertices and there are no vertices to the right of $\varphi^{-1}(|G|)$ in φ. Since $R(|G| - 1, \varphi, G) = \{\varphi^{-1}(|G|)\}$ and $L(|G| - 1, \varphi, G) = V_G - \{\varphi^{-1}(|G|)\}$, we conclude that the number of vertices in $L(|G| - 1, \varphi, G)$ that are adjacent to vertices in $R(|G| - 1, \varphi, G)$ is at least $\varepsilon(G)$; that is, $\delta(|G| - 1, \varphi, G) \geq \varepsilon(G)$. It follows from the fact $\mathrm{VS}(G) = \mathrm{VS}(\varphi, G) \geq \delta(|G| - 1, \varphi, G)$ that $\mathrm{VS}(G) \geq \varepsilon(G)$. To see that the inequality is tight if graph G has at least two vertices, consider the path graph $P_{|G|}$. By Lemma 2.3, graph $P_{|G|}$ has vertex separation 1, and $\varepsilon\left(P_{|G|}\right) = 1$. If G contains only one vertex, then G is the trivial graph, and hence $\varepsilon(G) = 0$ and $\mathrm{VS}(G) = 0$ (Lemma 2.2), and therefore the inequality is tight in this case as well, completing the proof. $\square$

Having proved a bound on the vertex separation of a graph in terms of its minimum vertex degree, we will shortly observe that it bears no reasonable functional relationship to $\Delta(G)$, the maximum degree of a vertex in G. In order to make this observation, we first derive the value for the vertex separation of the star graph.

Lemma 3.3 The vertex separation of the star graph S_n is 1 for any $n \geq 1$.

Proof. We prove the result by giving a layout φ with respect to vertex separation of star graph S_n such that φ has vertex separation 1. Star graph S_n contains $n + 1$ vertices, and since $n \geq 1$, it contains at least two vertices. Thus, Lemma 2.2 implies $\mathrm{VS}(S_n) > 0$. Figure 3.1 shows layout φ of S_n with vertex separation 1, where r is a vertex in S_n of degree n, and $r_1, \ldots, r_n$ are the vertices in S_n adjacent to r and having degree 1. Formally, we let $\varphi(r) = 1$ and $\varphi(r_i) = i + 1$ for all i,

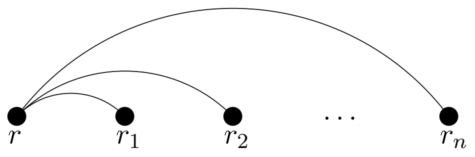

FIGURE 3.1: An optimal layout of star graph S_n with respect to vertex separation.

$1 \leq i \leq n$. It is clear that for any j, $1 \leq j \leq n+1$, the only vertex in set $L(j, \varphi, S_n)$ that is adjacent to a vertex in set $R(j, \varphi, S_n)$ is vertex r. Therefore, the vertex separation of layout φ is 1. It follows from the facts $\mathrm{VS}(S_n) > 0$, proved earlier, and $\mathrm{VS}(S_n) \leq \mathrm{VS}(\varphi, S_n)$ that $\mathrm{VS}(S_n) = 1$. □

Lemma 3.3 states that the vertex separation of star graph S_n is 1 regardless of the value of n. But $\Delta(S_n) = n$, and hence we conclude that the vertex separation of a general graph cannot be bounded from below by a function of $\Delta(G)$.

We now turn our attention to deriving lower and upper bounds on the cutwidth of graph G. The lower bound is in terms of the maximum degree $\Delta(G)$ of a vertex in G, and the upper bound is in terms of the number of vertices in G. We prove the bounds via a series of smaller results. First, we show that the cutwidth of any layout of graph G is bounded below by a function of $\Delta(G)$.

Claim 3.4 Given a layout φ of a graph G with maximum vertex degree $\Delta(G)$, the cutwidth of φ is at least $\left\lceil \frac{\Delta(G)}{2} \right\rceil$.

Proof. We derive the lower bound on cutwidth $\mathrm{CW}(\varphi, G)$ by finding a position in layout φ with edge cut equal to at least $\left\lceil \frac{\Delta(G)}{2} \right\rceil$. Consider a vertex u_Δ in G that has degree $\Delta(G)$; such a vertex must exist by the definition of $\Delta(G)$. There are $\Delta(G)$ edges incident to u_Δ, and therefore u_Δ is adjacent to at least $\left\lceil \frac{\Delta(G)}{2} \right\rceil$ vertices on one side of u_Δ in layout φ. More formally, vertex u_Δ is adjacent to at least $\left\lceil \frac{\Delta(G)}{2} \right\rceil$ vertices in either set $\widetilde{L}(\varphi(u_\Delta), \varphi, G) = \{\varphi^{-1}(1), \ldots, \varphi^{-1}(\varphi(u_\Delta) - 1)\}$ or set $R(\varphi(u_\Delta), \varphi, G) = \{\varphi^{-1}(\varphi(u_\Delta) + 1), \ldots, \varphi^{-1}(|G|)\}$; we denote this set by C. If $C = \widetilde{L}(\varphi(u_\Delta), \varphi, G)$, then $\theta(\varphi(u_\Delta) - 1, \varphi, G) \geq \left\lceil \frac{\Delta(G)}{2} \right\rceil$, since at least $\left\lceil \frac{\Delta(G)}{2} \right\rceil$ edges cross over position $\varphi(u_\Delta) - 1$ in the layout. If $C = R(\varphi(u_\Delta), \varphi, G)$, then $\theta(\varphi(u_\Delta), \varphi, G) \geq \left\lceil \frac{\Delta(G)}{2} \right\rceil$, because at least $\left\lceil \frac{\Delta(G)}{2} \right\rceil$ edges cross over position $\varphi(u_\Delta)$. We conclude that $\mathrm{CW}(\varphi, G) = \max_{1 \leq i \leq n} \theta(i, \varphi, G) \geq \left\lceil \frac{\Delta(G)}{2} \right\rceil$. □

Next, we use the lower bound just established to find an optimal layout of the star graph with respect to cutwidth.

Lemma 3.5 The cutwidth of the star graph S_n equals $\left\lceil \frac{n}{2} \right\rceil$ for any $n \geq 0$.

Proof. Because of Claim 3.4 and the fact $\Delta(S_n) = n$, we only need to demonstrate a layout φ of star graph S_n with cutwidth $\lceil n/2 \rceil$. We denote by r a vertex in S_n that has degree n, and by $r_1, \ldots, r_n$ the vertices that have degree 1 and are adjacent to vertex r. Figure 3.2 illustrates layout φ. Formally, we form layout φ

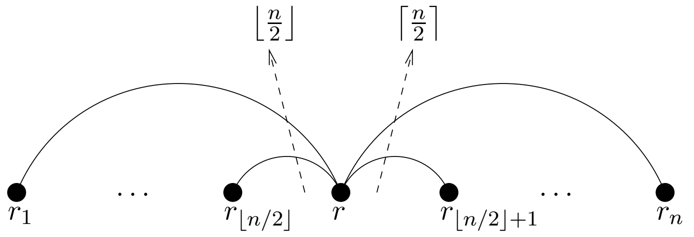
$\lfloor \frac{n}{2} \rfloor$
$\lceil \frac{n}{2} \rceil$
$\cdots$
$\cdots$
r_1
$r_{\lfloor n/2 \rfloor}$
r
$r_{\lfloor n/2 \rfloor + 1}$
r_n

FIGURE 3.2: An optimal layout of star graph S_n with respect to cutwidth. The numbers above the dashed arrows give the numbers of edges crossing the arrows.

such that it satisfies $\varphi(r_i) < \varphi(r) < \varphi(r_j)$ for all i and j such that $1 \leq i \leq \lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1 \leq j \leq n$. In other words, all vertices r_i satisfy $\varphi(r_i) = i$, all vertices r_j satisfy $\varphi(r_j) = j + 1$, and vertex r satisfies $\varphi(r) = \lfloor n/2 \rfloor + 1$.

We next consider the edge cut at each position of φ. The edge cut at position i' of layout φ such that $1 \leq i' \leq \lfloor n/2 \rfloor$ is at most $\lfloor n/2 \rfloor$, since $L(i', \varphi, S_n) \subseteq \left\{r_1, \ldots, r_{\lfloor n/2 \rfloor}\right\}$, and each vertex $r_{i'}$ in set $L(i', \varphi, S_n)$ is incident to exactly one edge, namely $r_{i'}u$. Similarly, the edge cut at position j' such that $\lfloor n/2 \rfloor + 2 \leq j' \leq n+1$ is at most $\lceil n/2 \rceil$, because $\lceil n/2 \rceil = n - \lfloor n/2 \rfloor$, and there are at most $\lceil n/2 \rceil$ vertices in set $R(j', \varphi, S_n)$, each of which is incident to exactly one edge $r_{j'}r$. Therefore, at most $\lceil n/2 \rceil$ edges cross over a position i of layout φ, where $1 \leq i \leq n+1$ and $i \neq \varphi(r) = \lfloor n/2 \rfloor + 1$. The edge cut at position $\varphi(r)$ is $\lceil n/2 \rceil$, because vertex r is adjacent to all vertices in $R(\varphi(r), \varphi, S_n) = \left\{r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n\right\}$, and no vertex in set $L(\varphi(r), \varphi, S_n) - \{r\}$ is adjacent to a vertex in set $R(\varphi(r), \varphi, S_n)$. Since the edge cut at each other position of layout φ was shown to be at most $\lceil n/2 \rceil$, we infer that $\mathrm{CW}(S_n) \leq \mathrm{CW}(\varphi, S_n) = \lceil n/2 \rceil$. The star graph S_n has $\Delta(S_n) = n$, and hence we conclude from Claim 3.4 that $\mathrm{CW}(S_n) = \lceil n/2 \rceil$. $\square$

We now derive the cutwidth of the complete graph, which will be used in proving an upper bound on the cutwidth of a general graph.

Lemma 3.6 The cutwidth of the complete graph K_n is $\left\lfloor \frac{n^2}{4} \right\rfloor$ for any $n \geq 0$.

Proof. We obtain the cutwidth of complete graph K_n by analyzing an arbitrary layout φ of K_n. We label the n vertices of K_n by $u_1, \ldots, u_n$ such that $\varphi(u_i) = i$. The edge cut at position 1 of φ is $n-1$, since vertex u_1 is adjacent to all $n-1$ vertices u_i such that $2 \leq i \leq n$. The edge cut at position 2 is $2(n-2)$, since vertex u_2 is adjacent to all $n-2$ vertices u_i such that $3 \leq i \leq n$, and vertex u_1, which comes before u_2 in φ, is also adjacent to all the $n-2$ vertices that are to the right of u_2 in φ; hence, $(n-2)+(n-2) = 2(n-2)$ edges cross over position 2 in φ.

In general, we consider an arbitrary position i of φ. There are $i-1$ vertices to the left of vertex $\varphi^{-1}(i)$ in φ. Each of these vertices is adjacent to the $n-i$ vertices to the right of $\varphi^{-1}(i)$. In addition, vertex u_i is adjacent to all $n-i$ vertices that are to the right of u_i. Hence, the edge cut at position i is $(i-1)(n-i)+(n-i) = i(n-i)$. This function achieves maximum values at $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Since $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, it follows that $\mathrm{CW}(\varphi, K_n) = \max_{1 \leq i \leq n} \theta(i, \varphi, K_n) = \lfloor n/2 \rfloor \lceil n/2 \rceil$. If n is even, $\mathrm{CW}(\varphi, K_n)$ is equal to $\frac{n^2}{4} = \left\lfloor \frac{n^2}{4} \right\rfloor$. If n is odd, then $\lfloor n/2 \rfloor \lceil n/2 \rceil = \left(\frac{n}{2} - \frac{1}{2}\right)\left(\frac{n}{2} + \frac{1}{2}\right) = \frac{n^2-1}{4} = \left\lfloor \frac{n^2}{4} \right\rfloor$, since $\frac{n^2-1}{4}$ is an integer and $\frac{n^2}{4} - \frac{n^2-1}{4} = \frac{1}{4} < 1$. Because layout φ was chosen arbitrarily, we see that every layout of K_n is an optimal layout, and hence $\mathrm{CW}(K_n) = \left\lfloor \frac{n^2}{4} \right\rfloor$. □

Finally, we give lower and upper bounds on the cutwidth of a graph as mentioned earlier.

Theorem 3.7 Any graph G satisfies $\left\lceil \frac{\Delta(G)}{2} \right\rceil \leq \mathrm{CW}(G) \leq \left\lfloor \frac{|G|^2}{4} \right\rfloor$, where $\Delta(G)$ is the maximum degree of a vertex in G. Furthermore, these inequalities are tight.

Proof. Inequality $\left\lceil \frac{\Delta(G)}{2} \right\rceil \leq \mathrm{CW}(G)$ follows from Claim 3.4, and by Lemma 3.5 it is tight. Inequality $\mathrm{CW}(G) \leq \left\lfloor \frac{|G|^2}{4} \right\rfloor$ is a consequence of Lemmas 2.1 and 3.6 and the fact that every graph is a subgraph of the complete graph $K_{|G|}$. Furthermore, Lemma 3.6 implies the inequality is tight. □

We remark that the cutwidth of graph G is trivially bounded above by the number of edges in G.

Having proved two simple bounds on the vertex separation and cutwidth of a general graph, we now turn our attention to computing the vertex separation of trees. In particular, we are interested in devising algorithms that update the vertex separation of a tree T as trees are added to or subtrees are removed from T. In the next chapter, we lay the groundwork for such algorithms.

Chapter 4

Vertex Separation of Trees (Preliminaries)

The goal of this chapter is to explain the concept of the vertex labelling of a tree, which encodes the relationship between the vertex separation and structure of the tree, including an optimal layout of the tree with respect to vertex separation. Vertex labelling is used in the algorithm by Ellis, Sudborough, and Turner [EST94], given at the end of the chapter, for computing the vertex separation of a tree. Our work on updating the vertex separation of a tree involves updating the tree's vertex labelling. Although the vertex labelling of a tree is defined only if the tree is rooted, while vertex separation and layout are defined on unrooted trees, it suffices to root the tree arbitrarily to make the necessary link between vertex labelling and vertex separation.

The concept of vertex labelling is used throughout the rest of the thesis, and we will introduce it in stages. Although all the material in this chapter is based on previous work [EST94], our exposition of it is different. In particular, we introduce the concept of a backbone of a tree, which is a useful construct for understanding the relationship between the vertex separation of a tree and the tree's structure.

This chapter is divided into two sections. In Section 4.1, we discuss results that are independent of whether the tree is rooted or not. The most important contributions of the section are the definition of a backbone of a tree and its properties. We also mention a few results from the original paper. In Section 4.2, we derive the concept of the vertex label and present an algorithm to compute the vertex separation of a rooted tree. The section contains many small results, mostly on the properties of a vertex label, that will be useful in later chapters.

4.1 Structure of a Tree and Its Vertex Separation

In this section, we investigate how the structure of an unrooted tree relates to the tree's vertex separation. We start by defining the main construct used in understanding this relationship.

Definition 4.1 An *m-backbone of a tree* T is a simple path in T whose branches all have vertex separation less than m, where m is a nonnegative integer. A $\mathrm{VS}(T)$-backbone of T is simply called a *backbone of* T.

To get a first taste of what an m-backbone is, we state and prove a simple consequence of the definition.

Lemma 4.1 If T is a nonempty tree, m is an integer satisfying $m > \mathrm{VS}(T)$, and u is any vertex in T, then the path $P = u$ is an m-backbone of T.

Proof. Every branch R of vertex u in tree T is a subgraph of T, and hence Lemma 2.1 implies $\mathrm{VS}(R) \leq \mathrm{VS}(T) < m$. Therefore, path $P = u$ is an m-backbone of tree T. □

The following result states for what values of m a tree has an m-backbone. Its proof is similar to the proof by Ellis, Sudborough, and Turner of Lemma 4.4.

Lemma 4.2 Each nonempty tree T has an m-backbone for any $m \geq \mathrm{VS}(T)$.

Proof. We first construct a backbone (that is, a $\mathrm{VS}(T)$-backbone) P of tree T. The lemma will then follow immediately, since all branches of P have vertex separation less than $\mathrm{VS}(T)$, and therefore they have vertex separation less than m for any $m \geq \mathrm{VS}(T)$; that is, path P is an m-backbone of T for any $m \geq \mathrm{VS}(T)$.

We construct path P by considering an optimal layout φ of T; that is, layout φ satisfies $\mathrm{VS}(\varphi, T) = \mathrm{VS}(T)$. If $\mathrm{VS}(T) = 0$, then it follows from Lemma 2.2 and the fact T is nonempty that T is the trivial tree. The only simple path in the trivial tree consists of its only vertex r_T. Vertex r_T does not have any branches, and hence the lemma follows trivially.

Therefore, assume $\mathrm{VS}(T) > 0$, so $n > 1$, where n is the number of vertices in tree T. Consider the path

$$P = sp(\varphi^{-1}(1), \varphi^{-1}(n)) \tag{4.1}$$

in T from the leftmost vertex $\varphi^{-1}(1)$ of layout φ to the rightmost vertex $\varphi^{-1}(n)$ of the layout. We demonstrate that each branch in $T[P]$ has vertex separation less than $\mathrm{VS}(T)$ by analyzing the vertex cut at position $\varphi(u)$ for each vertex u in T. We denote by T' the subgraph of T obtained by removing from tree T all edges incident to vertices on path P; that is, T' is a forest consisting of the trees in set $T[P]$. There are two cases in our argument, depending on whether or not u is a vertex in P.

First, we consider the vertex cut at position $\varphi(u)$ for any vertex u in tree T such that u is not on path P. Consider vertices u_1 and u_2 satisfying $\varphi(u_1) < \varphi(u) < \varphi(u_2)$, and such that the edge u_1u_2 is in P. Such an edge must exist, since u is not in P, $1 = \varphi\left(\varphi^{-1}(1)\right) < \varphi(u) < \varphi\left(\varphi^{-1}(n)\right) = n$ holds, and vertices $\varphi^{-1}(1)$ and $\varphi^{-1}(n)$ are both in P (Equation 4.1). Because vertex u_1 belongs to $L(\varphi(u), \varphi, T)$ and is adjacent to vertex u_2, which is in $R(\varphi(u), \varphi, T)$, removing from T all edges incident to u_1, which includes edge u_1u_2, reduces the vertex cut at position $\varphi(u)$ by at least one; this is because vertex u_1 becomes isolated, and thus is no longer adjacent to any vertex in $R(\varphi(u), \varphi, T)$. Furthermore, removing an arbitrary edge from tree T cannot increase the vertex cut at position $\varphi(u)$. Also, layout φ is a layout of graph T', since no vertices have been removed from T in constructing T'. Because vertex u_1 is in P, it is isolated in T', and the reasoning above implies $\delta(\varphi(u), \varphi, T') < \delta(\varphi(u), \varphi, T) \leq \max_{1 \leq i \leq n} \delta(i, \varphi, T) = \mathrm{VS}(T)$ for each vertex u not on path P.

Second, we consider the vertex cut at position $\varphi(u)$ for any vertex u in tree T such that u is on path P. If $u = \varphi^{-1}(n)$, then $\delta(\varphi(u), \varphi, T') = 0 < \mathrm{VS}(T)$, because the vertex cut at the nth position of a layout is always 0 and we assumed earlier that $\mathrm{VS}(T) > 0$. Hence, assume $u \neq \varphi^{-1}(n)$; that is, $\varphi(u) < n$. Then there is a vertex u_1 on path P that belongs to $L(\varphi(u), \varphi, T') = L(\varphi(u), \varphi, T)$, and such that u_1 is adjacent in tree T to a vertex u_2 in set $R(\varphi(u), \varphi, T') = R(\varphi(u), \varphi, T)$; if there were no such vertex, no edge in P would cross over position $\varphi(u)$, and hence vertex $\varphi^{-1}(n)$, which is in $R(\varphi(u), \varphi, T')$ by the assumption that $\varphi(u) < n$, could not be in P. We note that it is possible for vertex u_1 to be equal to u. But all edges incident to u_1 are removed when graph T' is constructed from tree T, and therefore vertex u_1 is no longer adjacent in T' to any vertex in set $R(i, \varphi, T')$; in particular, u_1 is not adjacent in T' to vertex u_2. We conclude that $\delta(\varphi(u), \varphi, T') < \delta(\varphi(u), \varphi, T) \leq \mathrm{VS}(T)$.

Finally, we combine the two cases considered in the previous two paragraphs and show that path P is a backbone of tree T. Combining the two cases, we see that $\delta(\varphi(u), \varphi, T') \leq \mathrm{VS}(T) - 1$ for any vertex u in T; that is, $\mathrm{VS}(T') \leq \mathrm{VS}(T) - 1$. Now consider a branch R of P in tree T. No vertex in R is in P, and therefore all edges in R are in T'. Hence, tree R is a subtree of T', and it follows from Lemma 2.1 that the vertex separation of R is at most $\mathrm{VS}(T') \leq \mathrm{VS}(T) - 1$. In conclusion, path P is a $\mathrm{VS}(T)$-backbone of T. □

Having shown that tree T has an m-backbone for any $m \geq \mathrm{VS}(T)$, we now prove that no m-backbone of T exists if $m < \mathrm{VS}(T)$. Taken together, these two results form the basis of understanding the relationship between the vertex separation and structure of a tree; the vertex labelling of a tree, introduced in the next section, essentially encodes backbones of the tree and its subtrees. The proof of the following result is adapted from the proof of the fact that if at most two branches of each vertex u in tree T have vertex separation k, and all other branches of u have vertex separation strictly less than k, then $\mathrm{VS}(T) \leq k$ [EST94].

Lemma 4.3 A tree T has no m-backbone for any integer m such that $1 \leq m < \mathrm{VS}(T)$.

Proof. We prove the lemma by contradiction; we assume tree T has an m-backbone, and then show that the m-backbone can be used to construct a layout of T with vertex separation $m < \mathrm{VS}(T)$. Assume tree T has an m-backbone $P = u_1, \ldots, u_p$ such that $m \geq 1$. We partition the branches of P in T into p sets $C_1, \ldots, C_p$ such that branch R of P belongs to C_i if and only if vertex u_i is adjacent in T to a vertex in R. Next, we impose an arbitrary order on the trees in each set C_i, denoting by φ_j^i an optimal layout of the jth tree in C_i, where $1 \leq i \leq p$ and $1 \leq j \leq |C_i|$. By Definition 4.1, each branch of m-backbone P has vertex separation at most $m-1$, and hence each layout φ_j^i has vertex separation at most $m-1$.

We now show how to construct a layout φ of T using layouts φ_j^i such that $\mathrm{VS}(\varphi, T) \leq m$. Figure 4.1 illustrates this construction. First, we describe the

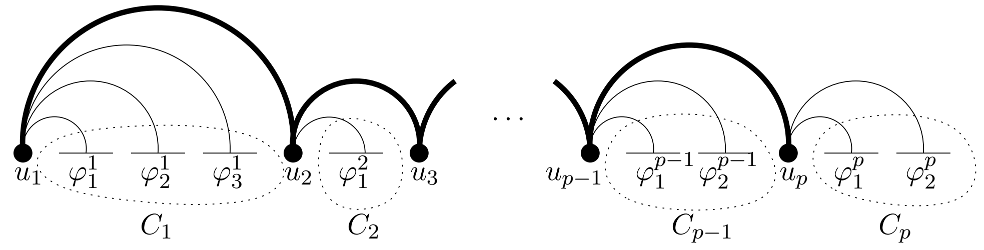

FIGURE 4.1: Constructing a layout of a tree from layouts of the branches of a path (the thick line) in the tree.

layout function φ for the vertices in path P. Intuitively, we place the vertices in P so that the number of positions strictly between positions $\varphi(u_{i+1})$ and $\varphi(u_i)$ is exactly equal to the number $n(i)$ of vertices in the trees in C_i. That is, we define $\varphi(u_1) = 1$ and $\varphi(u_{i+1}) = \varphi(u_i) + n(i) + 1$ for all i such that $1 \leq i \leq p-1$. Next, we assign the layout function φ to the vertices not in P. Intuitively, vertices in the trees in C_i are placed between vertices u_i and u_{i+1} in layout φ in the same order as in the layouts φ_j^i, and in such a way that if vertices v_1 and v_2 are consecutive in φ_j^i, then they are consecutive in φ. Formally, if v is a vertex in the jth tree in set C_i, then $\varphi(v) = \varphi(u_i) + \sum_{d=1}^{j-1} n(i,d) + \varphi_j^i(v)$, where $n(i,d)$ is the number of vertices in the dth tree in C_i; we note that $\sum_{d=1}^{|C_i|} n(i,d) = n(i)$.

Finally, we show that the vertex separation of layout φ is at most m by proving that the vertex cut at each position q of φ is at most m. We analyze two cases, depending on whether $\varphi^{-1}(q)$ is a vertex in a branch of path P or $\varphi^{-1}(q)$ is a vertex in P.

We first consider the case when $\varphi^{-1}(q)$ is a vertex in a branch of P. Suppose that q is a position in φ of a vertex in a branch of path P. Then q is a position of the part of φ corresponding to a layout φ_j^i; that is, q is such that $\varphi^{-1}(q)$ is a vertex in the jth tree in set C_i. From the definition of layout φ given in the previous paragraph, it follows that there is at most one vertex, u_i, outside of φ^i_j and satisfying $\varphi(u_i) \leq q$ that is adjacent to a vertex u_{i+1} in set $R(q, \varphi, T)$. Therefore, since the vertex separation of φ^i_j is at most $m - 1$ for all i and j such that $1 \leq i \leq p$ and $1 \leq j \leq |C_i|$, it follows that the vertex cut at position q is at most $(m-1)+1 = m$.

Next, we consider the case when $\varphi^{-1}(q)$ is a vertex on path P. Then the vertex cut at position q is at most 1, because the only vertex belonging to set $L(q, \varphi, T)$ that can be adjacent to a vertex in $R(q, \varphi, T)$ is u_i; the vertex to which u_i is adjacent in set $R(q, \varphi, T)$ are vertex u_{i+1} and the vertices in the trees in set C_i that are adjacent to u_i in tree T.

Combining the results of the previous two paragraphs, we get $\mathrm{VS}(\varphi, T) \leq \max\{m, 1\}$. Since $m \geq 1$ by assumption, it follows that $\mathrm{VS}(\varphi, T) \leq m < \mathrm{VS}(T)$, which is a contradiction. We conclude that tree T has no m-backbone for any integer m such that $1 \leq m < \mathrm{VS}(T)$. □

We remark that Lemma 4.3 is not valid for $m = 0$; hence the condition $m \geq 1$. To see this, consider the path graph P_n for any $n \geq 2$. Graph P_n is its own 0-backbone, because it does not have any branches, but it has vertex separation 1 according to Lemma 2.3. We also observe from Lemmas 4.1, 4.2, and 4.3 that finding an m-backbone of tree T is only interesting if $m = \mathrm{VS}(T)$.

We conclude the discussion on the relationship between the vertex separation and structure of a tree by stating a lemma that places a restriction on how many branches of a vertex u in tree T can have vertex separation equal to $\mathrm{VS}(T)$. This result is very helpful in concluding that a tree has vertex separation greater than m if a vertex in the tree has too many branches with vertex separation m.

Lemma 4.4 [EST94] If T is a tree and u is any vertex in T, then at most two subtrees in $T[u]$ have vertex separation $\mathrm{VS}(T)$, and all other subtrees in $T[u]$ have vertex separation strictly less than $\mathrm{VS}(T)$. □

In analyzing the running times of our algorithms, we will need a relationship between the number of vertices in a tree and its vertex separation. More specifically, we will need a lower bound in terms of k on the number of vertices in a tree with vertex separation k, and an upper bound on the vertex separation of a tree in terms of its size. The next two results give such bounds.

Lemma 4.5 [EST94] Given a positive integer k, the minimum number of vertices in a tree with vertex separation k is $\left(\frac{5}{6}\right) 3^k - \frac{1}{2}$. □

Theorem 4.6 [EST94] If T is a tree, then $\mathrm{VS}(T) = O(\lg |T|)$. Furthermore, there exists a tree T such that $\mathrm{VS}(T) = \Omega(\lg |T|)$. □

We note that given an integer k, there is no upper bound in terms of k on the number of vertices in a tree with vertex separation k. This follows easily from observing that the path graph P_n, which is a tree on n vertices, has vertex separation 1 for all $n \geq 2$ (Lemma 2.3).

In the next section, we use the results of this section to develop the concept of vertex labelling; in particular, we refine the concept of backbone to choose a unique backbone of a tree.

4.2 Vertex Labelling

In this section, we introduce the main concept used in this thesis: vertex labelling [EST94]. Our algorithms that update the vertex separation of a tree work by updating the vertex labelling of the tree. Vertex labelling is a generalization of vertex separation in the sense that the vertex separation of a tree T can be found trivially from the vertex labelling of T. In addition to the vertex separation, the vertex labelling encodes a backbone of tree T, backbones of the branches of the backbone, and so on in a recursive fashion. This information is sufficient for computing the vertex separation and an optimal layout of T recursively.

The current algorithm for computing the vertex labelling of a tree works on rooted trees [EST94]; we call the algorithm the Ellis-Sudborough-Turner algorithm. It does not matter which vertex is the root; rooting the tree corresponds to enforcing a particular recursive structure that is exploited by the algorithm. Finding a root of the tree so that the algorithm has the best possible running time seems not to have been investigated (see Chapter 8 for more on this point). We remark that the notions of vertex separation and optimal layout are independent of whether the tree is rooted or not. Therefore, the vertex separation of a rooted tree and the corresponding unrooted tree are the same. From this point on, we will assume all our trees are rooted at an arbitrary vertex.

The Ellis-Sudborough-Turner algorithm, the concepts of which are exploited in our work, finds the vertex labelling of tree T by computing the vertex labels of the children of the root r_T of T recursively, and then combining those labels to find the label of r_T. In order to understand our work, it is necessary to explain in detail the structure of the vertex label. Instead of just giving a formal definition, we first describe the concept that leads to the definition: the canonical backbone. This concept is implicit in the definition of vertex labelling, but is not explicitly recognized in the original work [EST94]. Our purpose for introducing the canonical backbone of a tree is twofold: first, it helps to understand the definition of the vertex label, and second, it is used explicitly in many of our arguments in conjunction with Lemmas 4.2 and 4.3, both of which are results on an m-backbone of a tree. The canonical backbone is essentially the shortest backbone of tree T that includes the root r_T of T, if possible; we will prove later that it is unique (Theorem 4.14).

Definition 4.2 The *canonical backbone of a tree* T, denoted by B_T, is the shortest backbone of T that contains the root r_T of T; if no such backbone exists, then B_T is simply the shortest backbone of T.

Before proceeding further, we need to show that the canonical backbone of a tree is well-defined, that is, that it is unique. We also mention that the orientation of a canonical backbone is irrelevant; that is, if P is the canonical backbone of a tree, then so is P^R. This property is invoked many times in our proofs, since when we reason about a canonical backbone, we usually argue in terms of the underlying oriented path.

Our proof of the uniqueness of the canonical backbone of a tree requires a new concept and a few preliminary results. The concept crucial to the proof, and also to most of the work that comes after it, is that of the criticality of a vertex.

Definition 4.3 [EST94] The *criticality of a vertex u in a tree T*, denoted by $crit(u, T)$, is the number of subtrees yielded by u that have vertex separation equal to $\mathrm{VS}(T)$. Vertex u is *critical* if its criticality is 2; otherwise, it is called *noncritical.*

We remark that the criticality of a vertex depends on the reference tree; for example, vertex u can be critical in a rooted subtree S of tree T, but it is noncritical in T if $\mathrm{VS}(S) < \mathrm{VS}(T)$, since no subtree of S can have vertex separation greater than $\mathrm{VS}(S)$ (Lemma 2.1).

The importance of the criticality concept is that if tree T contains a critical vertex u, then every backbone of T contains u; this will be shown shortly (Claim 4.8). Toward this goal, we state an important property of the criticality of a vertex in a tree. It is a simple consequence of Lemma 4.4.

Lemma 4.7 [EST94] If u is any vertex in a tree T, then $0 \leq \mathit{crit}(u, T) \leq 2$. Furthermore, there is at most one critical vertex in T. □

Later (Lemmas 4.16 and 4.17), we will see that tree T contains a critical vertex if and only if the canonical backbone B_T of T is nonmonotonic, and that the inflection vertex of B_T is the unique critical vertex in T. We recall that the inflection vertex of a nonmonotonic path P is denoted by inf_P. Hence, in anticipation of this result, we denote the critical vertex in tree T, if one exists, by inf_T.

When arguing about the critical vertex in tree T, or in general about any vertex u in T, it is often necessary to refer to the subtrees yielded by u that have vertex separation equal to $\mathrm{VS}(T)$. We therefore call a subtree R yielded by vertex u a *critical subtree of u in tree T* if $\mathrm{VS}(R) = \mathrm{VS}(T)$. The root of R is called a *critical child of vertex u in T*.

We mentioned earlier that every backbone of tree T contains the critical vertex inf_T in T, if there is one in T. We now prove this fact.

Claim 4.8 If a tree T has a critical vertex inf_T, then inf_T is in every backbone of T.

Proof. We prove the claim by showing that if P is a simple path in tree T not containing the critical vertex, then one of the branches of P has vertex separation at least $\mathrm{VS}(T)$. This implies by Definition 4.1 that path P is not a backbone of T. Since P is chosen arbitrarily so that it does not contain the critical vertex, we conclude that every backbone of T contains the critical vertex.

We consider an arbitrary simple path P in T not containing vertex inf_T. Then P is entirely contained in a branch of inf_T in tree T; this follows, since removing vertex inf_T and all its incident edges from T yields the branches of inf_T, but this operation does not disconnect the path P. Since inf_T is critical in T, it yields two critical subtrees R_1 and R_2, which are also branches of inf_T. Path P cannot be in both R_1 and R_2; say P is not in R_1. But then R_1 is a subtree of a branch R of P, and thus it follows from Lemma 2.1 and the fact that R_1 is a critical subtree of inf_T in tree T that $\mathrm{VS}(R) \geq \mathrm{VS}(R_1) = \mathrm{VS}(T)$, implying path P is not a backbone of tree T. □

Together with Claim 4.8, the next result will be important in inferring that the canonical backbone of tree T contains its root or its critical vertex.

Claim 4.9 If a tree T does not have a critical vertex, there is a backbone of T containing the root r_T of T.

Proof. We give a simple algorithm for constructing a simple path P in tree T that contains root r_T, and then prove that P is a backbone of T. The algorithm is an iterative one. Before entering the main loop, it starts with $P = r_T$. During one iteration, the algorithm does the following: if the right endpoint $last_P$ of P does not have a critical child, then the algorithm outputs path P and exits; otherwise, it adds a critical child of $last_P$ to the end of P, and then repeats the loop with the updated path P. We note that the algorithm needs to determine a critical child of vertex $last_P$. This can be done by computing the vertex separation of tree T and each subtree yielded by $last_P$, and then determining whether the vertex separation of one of the subtrees equals $\mathrm{VS}(T)$. The vertex separation of a tree is computable [EST94], and hence the algorithm is implementable; we are not concerned about time efficiency here.

We now prove that the simple algorithm just given computes a backbone of tree T. Clearly, path P returned by the algorithm contains root r_T and is monotonic, since it is constructed starting at r_T and at each step a child of the right endpoint of P is added to the end of P. This implies every branch R of P is a rooted subtree of T, and the root r_R of R is a child of a vertex in P. Since R is a subtree of T, it follows from Lemma 2.1 that $\mathrm{VS}(R) \leq \mathrm{VS}(T)$. Therefore, if $\mathrm{VS}(R) < \mathrm{VS}(T)$ holds for each branch R of P, then P is a backbone of T.

We prove $\mathrm{VS}(R) < \mathrm{VS}(T)$ holds for any branch R of path P by contradiction.

Assume to the contrary that $\mathrm{VS}(R) = \mathrm{VS}(T)$. Since the root r_R of R is a child of a vertex p in P, the fact that R is a critical subtree of p implies r_R is a critical child of p. The children of p are examined by the algorithm when the algorithm has already constructed the subpath $sp\,(r_T, p)$ of P. We now consider two cases, depending on whether p is the right endpoint of P or not.

First, we consider the case when p is the right endpoint of path P. Then $P = sp(r_T, p)$, that is, $p = last_P$, and the algorithm exits after examining the children of p, because p has no critical child. This contradicts the fact that r_R is a critical child of p. Therefore, the assumption $\mathrm{VS}(R) = \mathrm{VS}(T)$ must be false, and we conclude that $\mathrm{VS}(R) < \mathrm{VS}(T)$.

Second, we analyze the case when p is not the right endpoint of P. In this case, we consider the right neighbour $right(p, P)$ of p in P. The algorithm chooses vertex $right(p, P)$ to be added to the end of $sp(r_T, p)$ when it is examining the children of p. Hence, $right(p, P)$ is a critical child of p. However, r_R is also a critical child of p, and $r_R \neq right(p, P)$, since r_R is not in P. Lemma 4.7 therefore implies $crit(p, T) = 2$, and hence vertex p is critical in T. This contradicts the assumption that tree T does not have a critical vertex. We therefore conclude that $\mathrm{VS}(R) = \mathrm{VS}(T)$ cannot be true, and hence $\mathrm{VS}(R) < \mathrm{VS}(T)$. □

We see from Claims 4.8 and 4.9 that in order to prove the uniqueness of the canonical backbone of tree T, it suffices to show that the shortest backbone of all backbones of T that contain vertex s is unique, where $s = r_T$ if the canonical backbone contains r_T and $s = inf_T$ otherwise. However, we can prove a stronger result: assuming there is a backbone of T containing any vertex s in T, then the shortest backbone containing s is unique. When this result is combined with Claims 4.8 and 4.9, it implies the canonical backbone of T is unique, as we will show in detail in Theorem 4.14. The main idea behind the proof of the result is to apply induction on the minimum number of consecutive vertices shared among all shortest backbones of T containing s. The base case of the induction is trivial, because all shortest backbones of tree T that contain vertex s have at least one consecutive vertex in common, namely s.

We present our proof by means of a few claims. Our first result is used several times to conclude that a shorter subpath of a backbone is also a backbone if certain conditions are satisfied.

Claim 4.10 Consider a backbone $P = lpath(u,P) + u + rpath(u,P)$ such that the subpath $rpath(u,P)$ is nonempty.

1. If u is an interior vertex of P, that is, subpath $lpath(u,P)$ of P is nonempty as well, and $\mathrm{VS}\left(T[u]_{\mathit{left}(u,P)}\right) < \mathrm{VS}(T)$, then the subpath $u + rpath(u,P)$ of P is a backbone of T.

2. If u is the left endpoint of P, that is, subpath $lpath(u,P)$ is empty, and $\mathrm{VS}\left(T[u]_{\mathit{right}(u,P)}\right) < \mathrm{VS}(T)$, then the single-vertex subpath u of P is a backbone of T.

Proof. The claim is a direct consequence of Lemmas 2.5 and 2.4. We first observe that since path P is a backbone of tree T, all subtrees in set $T[P]$ of the branches of P in T have vertex separation less than $\mathrm{VS}(T)$. We prove each point in the claim separately.

We first consider the case when vertex u is an interior vertex of path P and $\mathrm{VS}\left(T[u]_{\mathit{left}(u,P)}\right) < \mathrm{VS}(T)$. Point 2 in Lemma 2.5 implies

$$T[u + rpath(u,P)] = (T[P] - T[lpath(u,P)]) \cup \left\{T[u]_{\mathit{left}(u,P)}\right\}.$$

Because all subtrees in $T[P]$ have vertex separation less than $\mathrm{VS}(T)$ and also $\mathrm{VS}\left(T[u]_{\mathit{left}(u,P)}\right) < \mathrm{VS}(T)$, we infer that all subtrees in set $T[u + rpath(u,P)]$ have vertex separation less than $\mathrm{VS}(T)$. We conclude that path $u + rpath(u,P)$ is a backbone of tree T.

We next consider the case when vertex u is the left endpoint of path P and $\mathrm{VS}\left(T[u]_{\mathit{right}(u,P)}\right) < \mathrm{VS}(T)$. Point 2 in Lemma 2.4 implies

$$T[u] = (T[P] - T[rpath(u,P)]) \cup \left\{T[u]_{\mathit{right}(u,P)}\right\}.$$

Like in the previous case, since all subtrees in set $T[P]$ have vertex separation less than $\mathrm{VS}(T)$ and $\mathrm{VS}\left(T[u]_{\mathit{left}(u,P)}\right) < \mathrm{VS}(T)$, it follows that all subtrees in set $T[u]$ have vertex separation less than $\mathrm{VS}(T)$. In conclusion, the path consisting of the single vertex u is a backbone of T. $\square$

As mentioned on page 39, the proof of the fact that the shortest backbone of T containing vertex s is unique proceeds by induction. We assume all shortest backbones containing s share at least $i-1$ consecutive vertices, and prove that all shortest backbones containing s share at least i consecutive vertices, where i is at most the number of vertices in all such shortest backbones. We also noted on page 39 that the base case $i = 1$ is trivial. However, for technical reasons, we also need to prove case $i = 2$ explicitly. The reason is that our proof of the induction step requires that a subpath on $i-1$ vertices of all shortest backbones containing s, guaranteed to exist by the induction hypothesis, has distinct endpoints; that is, the induction hypothesis requires that $i \geq 3$ so that $i - 1 \geq 2$. In the following claim, we therefore establish the case $i = 2$.

Claim 4.11 If there are at least two vertices in each shortest backbone of T containing vertex s, then all such shortest backbones share at least two consecutive vertices.

Proof. We prove the claim by showing that two arbitrary shortest backbones B_1 and B_2 of T containing vertex s share a vertex adjacent to s. This will imply all shortest backbones containing s have at least two consecutive vertices in common. Our proof is divided into two cases, depending on whether or not vertex s is an interior vertex of at least one backbone B_1 or B_2. The case when s is an interior vertex is further divided into two subcases. For both cases (and subcases), we assume to the contrary that all the neighbours of s in B_1 and B_2, which are all adjacent to vertex s in tree T, are distinct. We will derive the contradiction that B_2 is not a shortest backbone of T containing s.

We first consider the case when s is an interior vertex of at least one backbone B_1 or B_2. Since B_1 and B_2 were chosen arbitrarily, we may assume without loss of generality that s is an interior vertex of B_1. Then we can write $B_1 = \mathit{lpath}(s, B_1) + s + \mathit{rpath}(s, B_1)$, where $\mathit{lpath}(s, B_1)$ and $\mathit{rpath}(s, B_1)$ are both nonempty. Point 1 in Lemma 2.5 therefore yields

$$T[s] = (T[B_1] \cap T[s]) \cup \left\{T[s]_{\mathit{left}(s,B_1)}, T[s]_{\mathit{right}(s,B_1)}\right\}. \tag{4.2}$$

We next consider two subcases, depending on whether or not vertex s is an interior vertex of B_2.

We first analyze the subcase when s is an interior vertex of B_2. Then we can write $B_2 = \mathit{lpath}(s, B_2) + s + \mathit{rpath}(s, B_2)$, where $\mathit{lpath}(s, B_2)$ and $\mathit{rpath}(s, B_2)$ are both nonempty. Hence, point 1 in Lemma 2.5 implies the following analogue of Equation 4.2:

$$T[s] = (T[B_2] \cap T[s]) \cup \left\{T[s]_{\mathit{left}(s,B_2)}, T[s]_{\mathit{left}(s,B_2)}\right\}. \tag{4.3}$$

The neighbours $\mathit{left}(s, B_1)$, $\mathit{right}(s, B_1)$, $\mathit{left}(s, B_2)$, and $\mathit{right}(s, B_2)$ of s in B_1 and B_2 are all distinct; this is our original assumption. Hence, $T[s]_{\mathit{left}(s,B_2)} \neq T[s]_{\mathit{left}(s,B_1)}$ and $T[s]_{\mathit{left}(s,B_2)} \neq T[s]_{\mathit{right}(s,B_1)}$. Equating the right-hand sides of Equations 4.2 and 4.3, we therefore see that $T[s]_{\mathit{left}(s,B_2)} \in T[B_1] \cap T[s]$. Hence, subtree $T[s]_{\mathit{left}(s,B_2)}$ is a branch of backbone B_1, and thus $\mathrm{VS}\left(T[s]_{\mathit{left}(s,B_2)}\right) < \mathrm{VS}(T)$. Point 1 in Claim 4.10 now implies $s + \mathit{rpath}(s, B_2)$ is a backbone of T containing s that is strictly shorter than B_2.

We now analyze the subcase when vertex s is an endpoint of backbone B_2. Then we can write $B_2 = s + \mathit{rpath}(s, B_2)$ by orienting B_2 arbitrarily. Since every backbone containing s contains at least two vertices by assumption, point 1 in Lemma 2.4 implies

$$T[s] = (T[B_2] \cap T[s]) \cup \left\{T[s]_{\mathit{right}(s,B_2)}\right\}. \tag{4.4}$$

Since $\mathit{right}(s, B_2) \neq \mathit{left}(s, B_1)$ and $\mathit{right}(s, B_2) \neq \mathit{right}(s, B_1)$, which is our original assumption that the neighbours of s in both B_1 and B_2 are all distinct, we deduce that $T[s]_{\mathit{right}(s,B_2)} \neq T[s]_{\mathit{left}(s,B_1)}$ and $T[s]_{\mathit{right}(s,B_2)} \neq T[s]_{\mathit{right}(s,B_1)}$. It therefore follows from equating the right-hand sides of Equations 4.2 and 4.4 that $T[s]_{\mathit{right}(s,B_2)} \in T[B_1] \cap T[s]$. But B_1 is a backbone of tree T, and hence $\mathrm{VS}\left(T[s]_{\mathit{right}(s,B_2)}\right) < \mathrm{VS}(T)$. Point 2 in Claim 4.10 implies s is a backbone of T containing only one vertex, and therefore it is strictly shorter than backbone B_2, which contains at least two vertices.

Finally, we consider the case when vertex s is an endpoint of both backbones B_1 and B_2. We may assume without loss of generality that s is the left endpoint of both B_1 and B_2. Thus, we can write $B_1 = s + \mathit{rpath}(s, B_1)$ and $B_2 = s + \mathit{rpath}(s, B_2)$. Point 1 in Lemma 2.4 and the fact that every shortest backbone contains at least two vertices imply

$$T[s] = (T[B_1] \cap T[s]) \cup \left\{T[s]_{\mathit{right}(s,B_1)}\right\}, \tag{4.5}$$

and

$$T[s] = (T[B_2] \cap T[s]) \cup \left\{T[s]_{\mathit{right}(s,B_2)}\right\}. \tag{4.6}$$

Since $\mathit{right}(s, B_2) \neq \mathit{right}(s, B_1)$ by assumption, it follows that $T[s]_{\mathit{right}(s,B_2)} \neq T[s]_{\mathit{right}(s,B_1)}$. Equating the right-hand sides of Equations 4.5 and 4.6 implies that $T[s]_{\mathit{right}(s,B_2)} \in T[B_1] \cap T[s]$. Thus, $T[s]_{\mathit{right}(s,B_2)}$ is a branch of backbone B_1, and therefore $\mathrm{VS}\left(T[s]_{\mathit{right}(s,B_2)}\right) < \mathrm{VS}(T)$. It follows from point 2 in Claim 4.10 that s is a backbone of T that is strictly shorter than B_2.

In each case, we found a backbone of T containing vertex s that is shorter than backbone B_2. This is a contradiction, because B_2 was chosen to be a shortest backbone of T containing s. Hence, our original assumption, that all neighbours of s in both shortest backbones B_1 and B_2 are distinct, is false. We conclude that B_1 and B_2 share at least one neighbour of s, which is adjacent to s in tree T. Therefore, backbones B_1 and B_2 share at least two consecutive vertices, implying all shortest backbones of T containing s have at least two consecutive vertices in common. □

We now proceed to proving the induction step, which is divided into two claims in order to make it more manageable.

Claim 4.12 If the number of vertices in each shortest backbone of T containing vertex s is $\mathit{num} \geq 3$, and all such backbones share at least $q-1$ consecutive vertices $u_1, \ldots, u_{q-1}$, where $3 \leq q \leq \mathit{num}$, then either

1. u_1 is not an endpoint of any shortest backbone of T containing s, or
2. u_{q-1} is not an endpoint of any shortest backbone of T containing s.

Proof. We prove the claim by examining the endpoints u_1 and u_{q-1} of path $P = u_1, \ldots, u_{q-1}$. Path P is clearly a proper subpath of all shortest backbones containing s, since the number of vertices in P is strictly less than the number of vertices in every shortest backbone containing s ($\mathit{num} \geq q > q-1$). Furthermore, since $q \geq 3$, path P contains at least two vertices; that is, $u_1 \neq u_{q-1}$ holds. Therefore, vertices u_1 and u_{q-1} cannot both be endpoints of any shortest backbone of T containing vertex s.

Our goal is to show that if vertex u_1 is an endpoint of a shortest backbone B of T containing s, then u_1 is an endpoint of all shortest backbones containing s, and hence vertex u_{q-1} is not an endpoint of any shortest backbone containing s; this shows the second part of the claim. The case when vertex u_{q-1} is an endpoint of B is entirely symmetrical, since we can relabel the vertices $u_1, \ldots, u_{q-1}$ in reverse order.

We suppose that vertex u_1 is an endpoint of backbone B, and show that if there is a shortest backbone B_2 of T containing vertex s such that u_1 is not an endpoint of B_2, then B_2 is not a shortest backbone containing s. Assume u_1 is not an endpoint of B_2. Therefore, u_1 is an interior vertex of B_2, and we can write $B_2 = \mathit{lpath}(u_1, B_2) + u_1 + \mathit{rpath}(u_1, B_2)$. Point 1 in Lemma 2.5 implies

$$T[u_1] = (T[B_2] \cap T[u_1]) \cup \left\{ T[u_1]_{\mathit{left}(u_1,B_2)}, T[u_1]_{\mathit{right}(u_1,B_2)} \right\}. \tag{4.7}$$

We now consider backbone B. Since $\mathit{num} \geq 3$, B contains at least three vertices. Thus, by the assumption that vertex u_1 is an endpoint of B, we can write the backbone as $B = u_1 + \mathit{rpath}(u_1, B)$, where $\mathit{rpath}(u_1, B)$ is nonempty. It then follows from point 1 in Lemma 2.4 that

$$T[u_1] = (T\,[B] \cap T[u_1]) \cup \left\{ T[u_1]_{\mathit{right}(u_1,B)} \right\}. \tag{4.8}$$

We finally show how Equations 4.7 and 4.8 lead to the desired contradiction that B_2 is not a shortest backbone of T containing vertex s. Since $\mathit{left}(u_1, B_2) \neq \mathit{right}(u_1, B_2)$, at least one of the neighbours $\mathit{left}(u_1, B_2)$ and $\mathit{right}(u_1, B_2)$ does not equal $\mathit{right}(u_1, B)$. Because the orientation of backbone B_2 is not important, we can orient B_2 in such a way that the left neighbour $\mathit{left}(u_1, B_2)$ of u_1 in B_2 does not equal $\mathit{right}(u_1, B)$; we note that both $\mathit{left}(u_1, B_2)$ and $\mathit{right}(u_1, B_2)$ can differ from $\mathit{right}(u_1, B)$. By equating the right-hand sides of Equations 4.7 and 4.8, it follows that $T[u_1]_{\mathit{left}(u_1,B_2)} \in T[B] \cap T[u_1]$. Therefore, $T[u_1]_{\mathit{left}(u_1,B_2)}$ is a branch of backbone B, and hence $\mathrm{VS}\left(T[u_1]_{\mathit{left}(u_1,B_2)}\right) < \mathrm{VS}(T)$. Hence, it follows from point 1 in Claim

4.10 that path $u_1 + rpath(u_1, B_2)$ is a backbone of T. We will show shortly that $u_1+rpath(u_1, B_2)$ contains vertex s. Since $u_1+rpath(u_1, B_2)$ is strictly shorter than B_2, we have reached the contradiction to the fact that B_2 is a shortest backbone of T containing s. Therefore, if u_1 is an endpoint of backbone B, then it is an endpoint of backbone B_2.

It remains to show that path $u_1 + rpath(u_1, B_2)$ contains vertex s. Since $T[u_1]_{\mathit{left}(u_1,B_2)}$ is a branch of backbone B, which contains s, it follows $T[u_1]_{\mathit{left}(u_1,B_2)}$ does not contain s. Thus, the fact that $T[u_1]_{\mathit{left}(u_1,B_2)}$ contains subpath $lpath(u_1, B_2)$ of backbone B_2 implies $lpath(u_1, B_2)$ does not contain s. And since the path $B_2 = lpath(u_1, B_2) + u_1 + rpath(u_1, B_2)$ contains s, we conclude that path $u_1 + rpath(u_1, B_2)$ contains s. □

We next prove the second part of the induction step.

Claim 4.13 If the number of vertices in each shortest backbone of T containing s is $num \geq 3$, and all shortest backbones containing s have at least $q-1$ consecutive vertices in common, where $3 \leq q \leq num$, then all such shortest backbones have at least q consecutive vertices in common.

Proof. We prove the claim by considering two arbitrary shortest backbones of T containing s, and showing that if they share at least $q-1$ consecutive vertices, then there is at least one additional consecutive vertex that they have in common. We denote by P a path on any $q-1$ consecutive vertices $u_1, \ldots, u_{q-1}$ shared among all shortest backbones containing s. Path P is a subpath of each such shortest backbone. It follows from Claim 4.12 combined with the facts $num \geq 3$ and $3 \leq q \leq num$ that either u_1 is not an endpoint of any shortest backbone containing s, or u_{q-1} is not an endpoint of any shortest backbone containing s. Since we can orient path P arbitrarily, we may assume without loss of generality that vertex u_1 is not an endpoint of any shortest backbone of T containing vertex s.

We next show that there is a vertex z such that path $z + P$ is a subpath of all shortest backbones of T containing s, which proves the claim. Since $q \geq 3$, the number of vertices in P is $q-1 \geq 2$. Because path P is a subpath of all shortest backbones containing s, it follows that it is a subpath of any two such shortest backbones B_1 and B_2. Then vertex u_2, which is the neighbour of u_1 in P, is a neighbour of u_1 in both B_1 and B_2. Since the orientation of a backbone is not important, we orient backbones B_1 and B_2 so that u_2 is the right neighbour of u_1 in both B_1 and B_2; that is, $u_2 = \mathit{right}(u_1, B_1) = \mathit{right}(u_1, B_2)$.

Our goal now is to show that the left neighbours $\mathit{left}(u_1, B_1)$ and $\mathit{left}(u_1, B_2)$ of u_1 in B_1 and B_2, respectively, are equal, and therefore are the vertex z discussed in the previous paragraph; vertices $\mathit{left}(u_1, B_1)$ and $\mathit{left}(u_1, B_2)$ exist, since u_1 is not an endpoint of any shortest backbone containing s, and hence it is not an endpoint of either B_1 or B_2. Assume to the contrary that $\mathit{left}(u_1, B_1) \neq \mathit{left}(u_1, B_2)$. We will obtain the contradiction that B_2 is not a shortest backbone containing s. Point 1 in Lemma 2.4 implies

$$T[u_1] = (T[B_1] \cap T[u_1]) \cup \left\{T[u_1]_{\mathit{left}(u_1,B_1)}, T[u_1]_{\mathit{right}(u_1,B_1)}\right\}, \tag{4.9}$$

and

$$T[u_1] = (T[B_2] \cap T[u_1]) \cup \left\{T[u_1]_{\mathit{left}(u_1,B_2)}, T[u_1]_{\mathit{right}(u_1,B_2)}\right\}. \tag{4.10}$$

Since our assumption is $\mathit{left}(u_1, B_2) \neq \mathit{left}(u_1, B_1)$ and we know $\mathit{left}(u_1, B_2) \neq \mathit{right}(u_1, B_2) = \mathit{right}(u_1, B_1) = u_2$, we infer from equating the right-hand sides of Equations 4.9 and 4.10 that $T[u_1]_{\mathit{left}(u_1,B_2)} \in T[B_1] \cap T[u_1]$. Therefore, $T[u_1]_{\mathit{left}(u_1,B_2)}$ is a branch of backbone B_1, and hence $\mathrm{VS}\left(T[u_1]_{\mathit{left}(u_1,B_2)}\right) < \mathrm{VS}(T)$.

We conclude the proof by showing how the fact $\mathrm{VS}\left(T[u_1]_{\mathit{left}(u_1,B_2)}\right) < \mathrm{VS}(T)$ implies B_2 is not a shortest backbone of T containing vertex s. Since vertex u_1 is not an endpoint of backbone B_2, we can write $B_2 = \mathit{lpath}(u_1, B_2) + u_1 + \mathit{rpath}(u_1, B_2)$. Point 1 in Claim 4.10 thus implies path $u_1 + \mathit{rpath}(u_1, B_2)$ is a backbone of tree T. Because $T[u_1]_{\mathit{left}(u_1,B_2)}$ is a branch of backbone B_1, which contains vertex s, $T[u_1]_{\mathit{left}(u_1,B_2)}$ does not contain s, implying $\mathit{lpath}(u_1, B_2)$ does not contain s. The fact that B_2 contains vertex s implies path $u_1 + \mathit{rpath}(u_1, B_2)$ contains s, and therefore $u_1 + \mathit{rpath}(u_1, B_2)$ is a backbone of T containing s that is strictly shorter than B_2. This contradicts the fact that B_2 is a shortest backbone of T containing vertex s. Thus, $\mathit{left}(u_1, B_1) = \mathit{left}(u_1, B_2) = z$, and we see that path $z + P$ is a subpath of backbones B_1 and B_2. Since B_1 and B_2 are arbitrary, it follows that $z + P$ is a subpath of all shortest backbones of T containing vertex s. $\square$

We finally have all the essential results to be able to prove our goal that the canonical backbone of a tree is unique, thereby justifying its definition. The proof of this result is a combination of Claims 4.8, 4.9, 4.11, and 4.13.

Theorem 4.14 The canonical backbone B_T of a tree T is unique.

Proof. To prove the lemma, we first show that if tree T has no backbone that contains the root r_T of T, then all backbones of T share the critical vertex inf_T in T. Second, we use induction to prove that the shortest backbone of T containing vertex s is unique, where $s = r_T$ if there exists a backbone of T containing r_T and $s = inf_T$ if there is no backbone of tree T that contains root r_T. This will imply by Definition 4.2 that the canonical backbone of T is unique.

For the first part of the proof, suppose that tree T has no backbone that contains root r_T. Then by the contrapositive of Claim 4.9, there is a critical vertex inf_T in T. Claim 4.8 then implies inf_T is in every backbone of T.

We now prove that the shortest backbone of tree T containing vertex s is unique, where $s = r_T$ if there exists a backbone of T containing r_T and $s = inf_T$ if there is no backbone of T that contains root r_T. In other words, if there is a backbone of T that contains root r_T, we prove that the shortest backbone of all the backbones of T that contain r_T is unique. Similarly, if there is no such backbone, every backbone of T contains critical vertex inf_T, and we show that the shortest one is unique.

We prove this by induction on the minimum number of consecutive vertices common to all shortest backbones of T containing vertex s. That is, we show by induction that for all i such that $1 \leq i \leq num$, where num is the number of vertices in all shortest backbones containing s, all such shortest backbones share at least i consecutive vertices. The base case consists of subcases $i = 1$ and $i = 2$. Subcase $i = 1$ is trivially true, and Claim 4.11 implies subcase $i = 2$ is true. For the induction step, we assume all shortest backbones containing s contain at least $i - 1$ consecutive vertices in common, where $3 \leq i \leq num$; this is our induction hypothesis. Claim 4.13 implies all shortest backbones of T containing s share at least i consecutive vertices, which proves the induction step. If $i = num$, then all shortest backbones of T containing s share at least num consecutive vertices, and hence they must be identical, since each of them contains exactly num vertices. $\square$

For the purpose of computing the vertex separation and an optimal layout of tree T, we only consider canonical backbones. Theorem 4.14 implies this gives the algorithms well-defined, fixed behaviour. As mentioned at the beginning of this section, canonical backbones are implicitly encoded in the vertex labelling of tree T.

We now discuss vertex labels informally; we will supply a formal definition after motivating it. The label λ_u of a vertex u in tree T is a sequence of integers $(k_1, \ldots, k_p)$. The first integer in the sequence, k_1, equals the vertex separation of the subtree T_u of tree T rooted at u. The length p of λ_u indicates whether or not the root u of tree T_u is in the canonical backbone B_{T_u} of tree T_u. If vertex u is in B_{T_u}, then $p = 1$, and therefore $\lambda_u = (k_1)$. On the other hand, if u is not in backbone B_{T_u}, then $p > 1$. By Definition 4.2 and the contrapositive of Claim 4.9, T_u then has a critical vertex inf_{T_u}; furthermore, Claim 4.8 implies inf_{T_u} is in every backbone, including the canonical backbone of T_u, and therefore $\mathit{inf}_{T_u} \neq u$.

The remaining elements of label λ_u describe in a recursive fashion the subtree $T_u\langle v_1\rangle$ (Definition 2.1), where v_1 is the vertex in B_{T_u} closest to vertex u. Shortly (Lemma 4.15), we will show that canonical backbone B_{T_u} is nonmonotonic, implying by point 1 in Lemma 2.8 that v_1 is the inflection vertex of B_{T_u}. We will also show that $v_1 = \mathit{inf}_{T_u}$ (Lemma 4.16). The label of vertex u in tree $T_u\langle v_1\rangle$ is $(k_2, \ldots, k_p)$. Therefore, integer k_2 in label λ_u equals the vertex separation of tree $T_u\langle v_1\rangle$, and $p = 2$ if and only if vertex u is in the canonical backbone of $T_u\langle v_1\rangle$; otherwise, $p > 2$, and the remaining elements of label λ_u are determined recursively by considering the tree $T_u\langle v_1, v_2\rangle$, where v_2 is the vertex in the canonical backbone $B_{T_u\langle v_1\rangle}$ closest to u. We observe that since v_1 is the vertex in backbone B_{T_u} closest to vertex u, the tree $T_u\langle v_1\rangle$ is a branch of B_T in tree T_u. Therefore, $\mathrm{VS}(T_u\langle v_1\rangle) < \mathrm{VS}(T_u)$ holds (Definition 4.1). Since $k_1 = \mathrm{VS}(T_u)$ and $k_2 = \mathrm{VS}(T_u\langle v_1\rangle)$, we conclude that $k_1 > k_2$. Analogous reasoning can be used on tree $\mathrm{VS}(T_u\langle v_1, v_2\rangle)$ to infer that $k_2 > k_3$, and so on. That is, label λ_u is a strictly decreasing sequence of integers.

We now look more closely at the properties of vertex v_1. In the informal description of the label λ_u of vertex u, we chose v_1 to be the vertex in B_{T_u} closest to vertex u. This condition is rather cumbersome and also difficult to check. The following result, together with point 1 in Lemma 2.8, gives us a cleaner characterization of

v_1 as the inflection vertex of canonical backbone B_{T_u}.

Lemma 4.15 Given a tree T, consider the canonical backbone B_T of T. If backbone B_T does not contain the root of T, then B_T is nonmonotonic.

Proof. The proof proceeds by first assuming canonical backbone B_T is monotonic and does not contain root r_T, and then showing that B_T can be extended so that it does contain r_T; this is a contradiction. Since B_T is monotonic, point 2 in Lemma 2.8 implies one of its endpoints is closer to root r_T than is any other vertex in B_T. Because B_T can be oriented in an arbitrary way, we may assume without loss of generality that the left endpoint first_{B_T} of B_T is closer to r_T than is any other vertex in B_T; to reduce clutter, we simply denote first_{B_T} by first_T. We observe that no proper ancestor of first_T in T is in B_T, since every proper ancestor of first_T is closer to root r_T than vertex first_T. Hence, path $P = \mathit{sp}\,(r_T, z)$, where z is the parent of first_T, which consists only of ancestors of first_T, does not contain any vertex in B_T. We conclude that the path $P + B_T$ is simple.

We now show that $P + B_T = \mathit{sp}\,(r_T, z) + B_T$ is a backbone of tree T. By point 1 in Lemma 2.6, all branches of P, except branch $T[P]_{\mathit{first}_T}$, are subtrees of the branch $T[B_T]_z$ of B_T; that is, all branches in set $T[P] - \left\{T[P]_{\mathit{first}_T}\right\}$ are subtrees of $T[B_T]_z$. Since $T[B_T]_z$ is a branch of backbone B_T, it follows that $\mathrm{VS}\,(T[B_T]_z) < \mathrm{VS}(T)$. Lemma 2.1 therefore implies all subtrees in $T[P] - \left\{T[P]_{\mathit{first}_T}\right\}$ have vertex separation less than $\mathrm{VS}(T)$. Since point 3 in Lemma 2.6 implies

$$T[P + B_T] = (T[P] \cup T[B_T]) - \left\{T[P]_{\mathit{first}}, T[B_T]_{p_{\mathit{first}}}\right\},$$

the fact that all subtrees in $T[B_T]$ have vertex separation less than $\mathrm{VS}(T)$ lets us infer that all subtrees in set $T[P + B_T]$ have vertex separation less than $\mathrm{VS}(T)$. Thus, path $P + B_T$ is a backbone of tree T and contains root r_T. This contradicts the fact that B_T is the canonical backbone of T, because B_T contains r_T if there is a backbone of T that contains r_T (Definition 4.2). We conclude that if the canonical backbone B_T of tree T is monotonic, then it contains root r_T. Therefore, canonical backbone B_T is nonmonotonic. $\square$

Lemma 4.15 implies if the canonical backbone B_{T_u} of tree T_u is monotonic, then it contains the root u of T_u, and therefore the label of u is (k_1) according to our informal discussion. However, if B_{T_u} is nonmonotonic, then it may or may not contain vertex u.

We hinted earlier (the discussion immediately following Lemma 4.7) that the inflection vertex of a nonmonotonic canonical backbone B_T of tree T and the critical vertex in T are identical. That is why we chose the notation for the critical vertex of T to be inf_T, which is similar to notation inf_{B_T} for the inflection vertex of B_T. In the following two lemmas, we finally prove this assertion; that is, we prove the equivalence between the inflection and critical vertices.

Lemma 4.16 If a tree T has a nonmonotonic canonical backbone B_T, then it has a critical vertex inf_T; furthermore, $\mathit{inf}_{B_T} = \mathit{inf}_T$, where inf_{B_T} is the inflection vertex of B_T.

Proof. We prove the lemma by showing that the neighbours of inf_{B_T} in B_T are critical children of inf_{B_T} in tree T. To reduce clutter, we simply write B_T as B. Suppose that B is nonmonotonic with inflection vertex inf_B. Since inf_B is the inflection vertex of B, it is an interior vertex of B. Hence, we can write $B = \mathit{lpath}(\mathit{inf}_B, B) + \mathit{inf}_B + \mathit{rpath}(\mathit{inf}_B, B)$, where $\mathit{lpath}(\mathit{inf}_B, B)$ and $\mathit{rpath}(\mathit{inf}_B, B)$ are both nonempty. Consider the right endpoint *last* of $\mathit{lpath}(\mathit{inf}_B, B)$ and the left endpoint *first* of $\mathit{lpath}(\mathit{inf}_B, B)$. In other words, *last* and *first* are the left and right neighbours of inf_B in B, respectively. Since inf_B is the inflection vertex, both *last* and *first* are children of inf_B in tree T. We will show that $\mathrm{VS}\left(T_{\mathit{last}}\right) = \mathrm{VS}\left(T_{\mathit{first}}\right) = \mathrm{VS}(T)$, thus implying vertices *last* and *first* are critical children of inf_B. Hence, the criticality of inf_B in T is at least 2. By Lemma 4.7, the criticality of any vertex is at most 2. Thus, $\mathit{crit}(\mathit{inf}_B, T) = 2$, and vertex inf_B is critical in tree T (Definition 4.3).

We prove $\mathrm{VS}\left(T_{\mathit{last}}\right) = \mathrm{VS}\left(T_{\mathit{first}}\right) = \mathrm{VS}(T)$ by contradiction; we assume that $\mathrm{VS}\left(T_{\mathit{last}}\right) < \mathrm{VS}(T)$ or $\mathrm{VS}\left(T_{\mathit{first}}\right) < \mathrm{VS}(T)$, and show that a proper subpath of B is a backbone of tree T, contradicting the fact that backbone B is shortest. We only consider the case when $\mathrm{VS}\left(T_{\mathit{last}}\right) < \mathrm{VS}(T)$, since the case $\mathrm{VS}\left(T_{\mathit{first}}\right) < \mathrm{VS}(T)$ is symmetrical. By point 2 in Lemma 2.5, the following equation holds:

$$T[\mathit{inf}_B + \mathit{rpath}(\mathit{inf}_B, B)] = (T[B] - T[\mathit{lpath}(\mathit{inf}_B, B)]) \cup \{T[\mathit{inf}_B]_{\mathit{last}}\}\,. \qquad (4.11)$$

Tree $T[\mathit{inf}_B]_{\mathit{last}}$ is the branch of inf_B in T containing vertex *last* that is adjacent to inf_B in T; that is, it is the subtree yielded by inf_B and rooted at *last*, since *last* is a child of inf_B. In other words, $T[\mathit{inf}_B]_{\mathit{last}} = T_{\mathit{last}}$. The assumption is that $\mathrm{VS}\,(T_{\mathit{last}}) < \mathrm{VS}(T)$. Also, all branches in $T[B]$ have vertex separation less than $\mathrm{VS}(T)$, since they are branches of a backbone of T. We conclude from Equation 4.11 that all branches in $T[\mathit{inf}_B + \mathit{rpath}(\mathit{inf}_B, B)]$ have vertex separation less than $\mathrm{VS}(T)$, and hence $\mathit{inf}_B + \mathit{rpath}(\mathit{inf}_B, B)$ is a backbone of tree T. But path $\mathit{inf}_B + \mathit{rpath}(\mathit{inf}_B, B)$ is strictly shorter than B, contradicting the fact that B is the shortest backbone containing inf_B. Therefore, $\mathrm{VS}\,(T_{\mathit{last}}) = \mathrm{VS}(T)$ must hold, since Lemma 2.1 implies $\mathrm{VS}\,(T_{\mathit{last}}) > \mathrm{VS}(T)$ is impossible. □

We now prove the converse of Lemma 4.16, thus establishing the equivalence between the inflection vertex of a nonmonotonic canonical backbone of tree T and the critical vertex in T.

Lemma 4.17 If a tree T has a critical vertex inf_T, then the canonical backbone B_T is nonmonotonic; furthermore, $\mathit{inf}_T = \mathit{inf}_{B_T}$.

Proof. We prove the lemma by showing that the critical children of inf_T must be in B_T. Suppose tree T has a critical vertex inf_T. Then inf_T yields two subtrees R_1 and R_2 of T such that

$$\mathrm{VS}(R_1) = \mathrm{VS}(R_2) = \mathrm{VS}(T). \tag{4.12}$$

Also, it follows from Claim 4.8 that vertex inf_T is in every backbone of T; in particular, inf_T is in B_T. We will show shortly that the roots r_{R_1} and r_{R_2} of trees R_1 and R_2, respectively, must be in B_T, implying path $r_{R_1} + \mathit{inf}_T + r_{R_2}$ is a subpath of B_T. But path $r_{R_1} + \mathit{inf}_T + r_{R_2}$ is nonmonotonic with inflection vertex inf_T, and therefore it follows from Lemma 2.9 that canonical backbone B_T is nonmonotonic with inflection vertex inf_T.

It remains to show that vertices r_{R_1} and r_{R_2} are in canonical backbone B_T. We prove this by contradiction. Assume r_{R_1} is not in B_T; the case when r_{R_2} is not in B_T is identical, except that subscript 2 is used in place of subscript 1. We derive the contradiction that canonical backbone B_T is not a backbone of tree T. The simple path from vertex inf_T to any vertex in subtree R_1 is unique and contains vertex r_{R_1}. Since our assumption is that r_{R_1} is not in B_T, it follows that no vertex in R_1 is in B_T. Hence, R_1 is a subtree of a branch R of backbone B_T, and hence we infer from Lemma 2.1 and Equation 4.12 that $\text{VS}(R) \geq \text{VS}(T)$, which contradicts the fact that B_T is a backbone of tree T. Therefore, the assumption that r_{R_1} is not in canonical backbone B_T must be false. Hence, vertex r_{R_1} is in B_T, and by the same argument, vertex r_{R_2} is in B_T as well. □

Lemmas 4.16 and 4.17 together imply the canonical backbone B_{T_u} of tree T_u is nonmonotonic if and only if T_u has a critical vertex. If T_u does not have a critical vertex, then B_{T_u} is monotonic, and therefore by Lemma 4.15 it contains the root u of T_u. Hence, the label λ_u of vertex u is (k_1) according to our informal discussion of the vertex label. If canonical backbone B_{T_u} is nonmonotonic, then $\lambda_u = (k_1)$ if and only if B_{T_u} contains vertex u; otherwise, $p > 1$, and we derive the remaining integers in label λ_u by considering the subtree $T_u\langle v_1\rangle$, where $v_1 = \mathit{inf}_{T_u}$. This correspondence between the critical vertex in tree T and the inflection vertex of the canonical backbone of T is not made in the original paper [EST94]. We believe it makes the formal definition of the vertex label easier to understand on an intuitive level.

As discussed starting on page 48, the label $\lambda_u = (k_1, \ldots, k_p)$ is a sequence of integers. However, in computing with λ_u, we also need to know the criticalities of the vertices $v_1, \ldots, v_{p-1}, u$ in trees $T_u\langle\rangle = T_u, T_u\langle v_1\rangle, \ldots, T_u\langle v_1, \ldots, v_{p-1}\rangle$, respectively. Vertices $v_1, \ldots, v_{p-1}$ are all critical in trees $T_u, \ldots, T_u\langle v_1, \ldots, v_{p-2}\rangle$, respectively, and hence their ciritcality is 2. But the criticality of vertex u in tree $T_u\langle v_1, \ldots, v_{p-1}\rangle$ can be 0, 1, or 2. Therefore, in most instances, we need both the sequence of integers and the criticality of u. The formal definition of the vertex label takes this extra information into account.

Definition 4.4 [EST94] The *label of a vertex u in a rooted tree T*, denoted $\lambda_{u,T}$ or simply λ_u if T is clear from context, is a sequence of integers $(k_1, \ldots, k_p)_c$ together with an integer c for which there exists a sequence of vertices $v_1, \ldots, v_p$ in subtree T_u of T such that the following three conditions are satisfied:

1. $\text{VS}(T_u) = k_1$;

2. for $1 \leq i \leq p-1$, vertex v_i is critical in tree $T_u\langle v_1, \ldots, v_{i-1}\rangle$ and we have $\mathrm{VS}(T_u\langle v_1, \ldots, v_i\rangle) = k_{i+1}$;

3. $v_p = u$, and either vertex u is critical or there is no critical vertex in tree $\mathrm{VS}(T_u\langle v_1, \ldots, v_{p-1}\rangle)$.

Integer c, also denoted by $\mathit{crit}(\lambda_u, T)$ or $\mathit{crit}(\lambda_u)$, equals the criticality of vertex u in tree $\mathrm{VS}(T_u\langle v_1, \ldots, v_{p-1}\rangle)$ and is called the *criticality of* λ_u. Label λ_u is *critical* or *noncritical* depending on whether $c = 2$ or $c \neq 2$, respectively.

Lemma 4.18 [EST94] The label of each vertex in a rooted tree is unique. □

The consistency of Definition 4.4 with the informal discussion leading up to the definition can be easily seen once we identify the vertex v_i, which is critical in tree $T_u\langle v_1, \ldots, v_{i-1}\rangle$ if $i < p$, with the inflection vertex of the canonical backbone $B_{T_u\langle v_1, \ldots, v_{i-1}\rangle}$ of $T_u\langle v_1, \ldots, v_{i-1}\rangle$.

The definition of the vertex label is not simple, and therefore we state a few lemmas by Ellis, Sudborough, and Turner [EST94] that justify claims made informally earlier.

Lemma 4.19 [EST94] The vertex separation of a tree is equal to the first integer in the label of the root of the tree. □

Lemma 4.20 [EST94] A vertex label $(k_1, \ldots, k_p)_c$ is a strictly decreasing sequence; that is, $k_1 > \cdots > k_p \geq 0$. □

Lemma 4.21 [EST94] There is no critical vertex in the subtree T_u of a tree T or vertex u is critical in T_u if and only if the label of u is $(\mathrm{VS}(T_u))_{\mathit{crit}(u, T_u)}$. □

Lemma 4.22 [EST94] If $\lambda_u = (k_1, \ldots, k_p)_c$ is the label of a vertex u in a tree T with $p > 1$, then $(k_2, \ldots, k_p)_c$ is the label of u in tree $T_u\left\langle \mathit{inf}_{T_u}\right\rangle$. □

Since most of the results in later chapters involve reasoning about vertex labels, it is convenient to introduce additional notation and terminology. We define the *label of tree* T as the label of the root of T, and denote it by λ_T. The *vertex labelling of* T is a collection of the labels of all vertices in T; it is denoted by Λ_T. We say that two vertex labels $\lambda = (k_1, \ldots, k_p)_c$ and $\lambda' = \left(k'_1, \ldots, k'_{p'}\right)_{c'}$ are *equal up to criticality* if $p = p'$ and $k_i = k'_i$ for all i, $1 \leq i \leq p = p'$. Labels λ and λ' are said to be *equal* if they are equal up to criticality and $c = c'$.

The *length of label* $\lambda_u = (k_1, \ldots, k_p)$ is defined to be equal to p, and is denoted by $|\lambda_u|$. We now make a few simple observations about unit-length vertex labels.

Lemma 4.23 If λ_u is the label of a vertex u in a tree T such that $|\lambda_u| = 1$, and w is a vertex in the subtree T_u of T such that $w \neq u$, then w is noncritical in T_u.

Proof. If vertex w is critical in tree T_u, then it follows from Lemma 4.21 and the fact $w \neq u$ that $|\lambda_u| > 1$. This contradicts the supposition that $|\lambda_u| = 1$. □

Lemma 4.24 If λ_u is the label of a vertex u in a tree T such that $|\lambda_u| = 1$ and is noncritical, then there is no critical vertex in the subtree T_u of T.

Proof. Lemma 4.21 implies the criticality of label λ_u equals the criticality of vertex u in tree T_u. Hence, since λ_u is noncritical, so is u. If w is a vertex in T_u such that $w \neq u$, then by Lemma 4.23 w is noncritical. Therefore, there is no critical vertex in tree T_u. □

In order to derive running times of algorithms that compute with vertex labels, it is necessary to have an upper bound on the length of a vertex label. The following result is a simple consequence of Theorem 4.6 and Lemmas 2.1, 4.19, and 4.20.

Lemma 4.25 [EST94] The length of the label of any vertex in a tree T is at most $\mathrm{VS}(T) + 1 = O(\lg |T|)$. Furthermore, there exists a tree T such that the label of the root of T has length $\mathrm{VS}(T) + 1 = \Omega(\lg |T|)$. □

An integer k_i in label $\lambda_u = (k_1, \ldots, k_p)_c$ is also called the *ith element of label* λ_u, or simply an *element of* λ_u if its position in the label is unimportant; it is denoted by $\lambda_u(i)$. The *last element of label* λ_u is k_p. Vertex v_i, as used in Definition 4.4, and element k_i of λ_u are said to *correspond to each other.* We call the tree $T\langle v_1, \ldots, v_{i-1}\rangle_{v_i}$ the *subtree corresponding to element* k_i. In the following two lemmas, we simply restate aspects of Definition 4.4 using the newly introduced notation and terminology.

Lemma 4.26 Given the label λ_u of a vertex u in a tree T, consider the sequence $v_1, \ldots, v_{|\lambda_u|}$ of vertices corresponding to elements $\lambda_u(1), \ldots, \lambda_u(|\lambda_u|)$ of λ_u, respectively. Then it follows that $\mathrm{VS}(T_u\langle v_1, \ldots, v_i\rangle) = \lambda_u(i+1)$, where $1 \leq i \leq |\lambda_u| - 1$.

Proof. Letting $k_i = \lambda_u(i)$, we see readily that the lemma is equivalent to condition 2 in Definition 4.4. □

Lemma 4.27 The vertex corresponding to the last element of the label λ_u of a vertex u in a tree T is u. Furthermore, u does not correspond to any element of λ_u that is not the last element.

Proof. The first part of the lemma is obvious (condition 3 in Definition 4.4). We justify the second part by observing that if $v_i = u$ such that $i < p = |\lambda_u|$, then tree $T_u\langle v_1, \ldots, v_i\rangle$ is empty. Since $i < p$, there is an element k_{i+1} of λ_u that corresponds to a vertex v_{i+1} in $T_u\langle v_1, \ldots, v_i\rangle$. But tree $T_u\langle v_1, \ldots, v_i\rangle$ is empty, and therefore v_{i+1} is not in $T_u\langle v_1, \ldots, v_i\rangle$, which is a contradiction. We conclude that $v_i \neq u$. □

Element k_i of label λ_u is said to be *critical* or *noncritical* depending on whether the vertex v_i corresponding to k_i is critical or noncritical in tree $T_u\langle v_1, \ldots, v_{i-1}\rangle$, respectively. When we argue about vertex v_i being critical or noncritical, we frequently argue in terms of the corresponding element k_i. The following three simple lemmas make the necessary connection between v_i and k_i.

Lemma 4.28 If an element k_i of the label λ_u of a vertex u in a tree T is noncritical, then the vertex corresponding to k_i is u, and k_i is the last element of λ_u.

Proof. Since k_i is a noncritical element, the vertex v_i corresponding to k_i is noncritical in tree $T_u\langle v_1, \ldots, v_{i-1}\rangle$. But then $v_i = u$, and k_i is the last element of label u, because only the last vertex in the sequence $v_1, \ldots, v_p$ of vertices may be noncritical in tree $T_u\langle v_1, \ldots, v_{p-1}\rangle$, and the last vertex in the sequence is u (conditions 2 and 3 in Definition 4.4, respectively). □

Lemma 4.29 If the first element of a label λ_u of a vertex u in a tree T is noncritical, then $|\lambda_u| = 1$.

Proof. Since k_1 is a noncritical element of label λ_u, then Lemma 4.28 implies the vertex corresponding to k_1 is u. From the contrapositive of Lemma 4.27, it follows that k_1 is the last element of λ_u. Because k_1 is both the first and last element of label λ_u, it follows that it is the only element of λ_u. $\square$

Lemma 4.30 If λ_u is the label of a vertex u in a tree T such that $|\lambda_u| > 1$, then $\lambda_u(i)$ is a critical element of λ_u for all i such that $1 \leq i \leq |\lambda_u| - 1$.

Proof. This lemma essentially restates the second condition in the definition of the vertex label (Definition 4.4): vertex v_i is critical in tree $T_u\langle v_1, \ldots, v_{i-1}\rangle$ for all i such that $1 \leq i \leq p - 1 = |\lambda_u| - 1$. The conclusion follows by observing that the element of label λ_u corresponding to vertex v_i is $\lambda_u(i)$. $\square$

We next introduce the operation of adding an element to the beginning of a vertex label. This operation is used later in this section in computing the label of a vertex u in tree T from the labels of the children of u. Intuitively, adding an element k to label u corresponds to attaching a tree S with vertex separation $k > \mathrm{VS}(T)$ whose root is critical in S. Lemma 4.31 supports this intuition. Given a label $\lambda_u = (k_1, \ldots, k_p)_c$ and integer k such that $k > k_1$, label $(k, k_1, \ldots, k_p)_c$ is said to be the label λ_u *prepended with* k. The following lemma gives an indication of when the prepending operation is useful.

Lemma 4.31 Given trees T and S such that $\mathrm{VS}(T) < \mathrm{VS}(S)$, if the root r_S of S is critical in S, then the label of tree $T \vdash_u S$ is the label of T prepended with $\mathrm{VS}(S)$, where u is an arbitrary vertex in T and $T \vdash_u S$ is the tree with root r_T formed from T and S by adding the edge ur_S.

Proof. The lemma is a consequence of the recursive definition of the vertex label. We first show that the vertex separation of tree $T \vdash_u S$ equals $\mathrm{VS}(S)$. Then we show that vertex r_S is critical in tree $T \vdash_u S$, and use Lemma 4.22 to derive the label of tree T from the label of $T \vdash_u S$.

We first show that $\mathrm{VS}(T \vdash_u S) = \mathrm{VS}(S)$. Since vertex r_S is critical in tree S, Lemma 4.17 implies the canonical backbone B_S of S contains r_S. And because root r_S is attached to vertex u in constructing tree $T \vdash_u S$, it follows that T is a branch of B_S in $T \vdash_u S$; that is, $(T \vdash_u S)[B_S] = S[B_S] \cup \{T\}$. It follows from the fact that B_S is a backbone of tree S that every subtree in set $S[B_S]$ has vertex separation less than VS(S). By assumption, we know that VS(T) < VS(S). Hence, each tree in set $(T \vdash_u S)[B_S]$ has vertex separation less than VS(S), and thus B_S is a VS(S)-backbone of $T \vdash_u S$. Lemma 4.3 therefore implies VS($T \vdash_u S$) $\leq$ VS(S). It follows from Lemma 2.1 and the fact that tree S is a subtree of tree $T \vdash_u S$ that VS($T \vdash_u S$) $\geq$ VS(S). We conclude that VS($T \vdash_u S$) = VS(S).

We next show that vertex r_S is critical in tree $T \vdash_u S$. Root r_S is critical in tree S, and hence it yields two subtrees R_1 and R_2 in S each with vertex separation VS(S). Vertex r_S clearly yields subtrees R_1 and R_2 in tree $T \vdash_u S$, and since VS($T \vdash_u S$) = VS(S), it follows that r_S is critical in $T \vdash_u S$. Since $r_T \neq r_S$, the label $\lambda_{T \vdash_u S}$ of the root r_T is $(k_1, \ldots, k_p)_{crit(\lambda_{T \vdash_u S})}$, where $p > 1$ (Lemma 4.21) and $k_1 = \text{VS}(T \vdash_u S) = \text{VS}(S)$ (Lemma 4.19). And because $(T \vdash_u S)\langle r_S \rangle = T$, Lemma 4.22 implies the label λ_T of tree T is $(k_2, \ldots, k_p)_{crit(\lambda_{T \vdash_u S})}$. Thus, label $\lambda_{T \vdash_u S}$ is the label λ_T prepended with $k_1 = \text{VS}(S)$. □

We now rigorously tie together the concepts of the vertex label and canonical backbone. Our algorithms work with the vertex labellings of trees, but their correctness proofs frequently employ canonical backbones. We first investigate the relationship between the length of the label of a tree and when the canonical backbone of the tree contains the root of the tree.

Lemma 4.32 Given a tree T with label λ_T and root r_T, the canonical backbone B_T of T contains r_T if and only if $|\lambda_T| = 1$. Furthermore, root r_T is an endpoint of B_T if and only if $|\lambda_T| = 1$ and $crit(\lambda_T) < 2$.

Proof. We first show that canonical backbone B_T contains root r_T if and only if $|\lambda_T| = 1$. First, suppose B_T contains r_T. We consider two cases, depending on whether or not tree T has a critical vertex. If tree T does not have a critical vertex, then Lemma 4.21 implies $\lambda_T = (\text{VS}(T))_{crit(r_T,T)}$, and hence $|\lambda_T| = 1$. If there is a critical vertex inf_T in T, then Lemma 4.17 implies backbone B_T is nonmonotonic and the inflection vertex of B_T equals inf_T. Since by point 1 in Lemma 2.8 the inflection vertex of B_T is closer to root r_T than any other vertex in B_T, and because backbone B_T contains r_T, the inflection vertex of B_T is $r_T = inf_T$. Thus, root r_T is critical in tree T, and therefore Lemma 4.21 yields $\lambda_T = (\mathrm{VS}(T))_{crit(r_T,T)}$, implying that $|\lambda_T| = 1$. Hence, in both cases, $|\lambda_T| = 1$ holds.

Having proved the forward implication, we now show the backward implication. Suppose that $|\lambda_T| = 1$. If there is not a critical vertex in tree T, then it follows from Claim 4.9 that there is a backbone of T containing root r_T, and therefore canonical backbone B_T contains r_T. If there is a critical vertex inf_T in tree T, then Lemma 4.23 implies $inf_T = r_T$. Since by Claim 4.8 the critical vertex in T is in every backbone of T, backbone B_T contains inf_T. Thus, in both cases, canonical backbone B_T contains root r_T.

We next show that root r_T is an endpoint of canonical backbone B_T if and only if $|\lambda_T| = 1$ and $crit(\lambda_T) < 2$. First, suppose that r_T is an endpoint of B_T. Since B_T contains r_T, point 1 in Lemma 2.8 implies if B_T is nonmonotonic, then r_T is the inflection vertex of B_T, which contradicts the assumption that it is an endpoint of B_T. Hence, B_T is monotonic. The contrapositive of Lemma 4.17 then implies tree T does not have a critical vertex. It therefore follows from Lemma 4.21 that $\lambda_T = (\mathrm{VS}(T))_{crit(r_T,T)}$, and therefore $|\lambda_T| = 1$ holds, and the criticality of label λ_T equals the criticality of root r_T in tree T. Since there is no critical vertex in T, it follows from Lemma 4.7 that $crit(r_T, T) < 2$, and therefore $crit(\lambda_T) < 2$.

We conclude the proof by showing the backward implication of the second part of the lemma. Suppose that $|\lambda_T| = 1$ and $crit(\lambda_T) < 2$. Lemma 4.24 implies there is no critical vertex in tree T. Therefore, it follows from Claim 4.9 that there is a backbone of T containing root r_T, and hence canonical backbone B_T contains r_T. By the contrapositive of Lemma 4.16, backbone B_T is monotonic. We conclude that root r_T is an endpoint of B_T; if r_T were not an endpoint of B_T, then backbone B_T would be nonmonotonic. □

The next result describes the canonical backbone of a tree when the label of the tree has at least two elements.

Lemma 4.33 If T is a tree with label λ_T such that $|\lambda_T| > 1$, and v_1 is the vertex in T corresponding to the first element of λ_T, then

1. the canonical backbone B_T of T is nonmonotonic,

2. v_1 is the inflection vertex of B_T, and

3. one of the branches of B_T in T is the tree $T\langle v_1 \rangle$.

Proof. The lemma is a simple consequence of results we have already established. Suppose that $|\lambda_T| > 1$. Then Lemma 4.30 implies the first element $\lambda_T(1)$ is a critical element. Hence, the vertex v_1 corresponding to $\lambda(1)$ is critical in tree T. It therefore follows from Lemma 4.17 that canonical backbone B_T is nonmonotonic, and the inflection vertex of B_T is v_1. Since vertex v_1 is closer to root r_T than any other vertex in B_T (point 1 in Lemma 2.8), we conclude that tree $T\langle v_1 \rangle$ contains the parent of v_1; vertex v_1 has a parent, because Lemma 4.27 yields $v_1 \neq r_T$. Therefore, $T\langle v_1 \rangle$ is a branch of canonical backbone B_T in tree T. □

Having covered basic results about the structure and properties of the vertex label, we now turn to the problem of computing the labels for all vertices in tree T. As noted earlier (Lemma 4.19), the label of the root of T gives us the vertex separation of tree T. The vertex labelling of tree T can also be used to compute an optimal layout of T in $O(|T| \lg |T|)$ time [EST94]. We remark that Skodinis gave an $O(|T|)$ algorithm to compute both the vertex separation and an optimal layout of tree T [Sko00]. His algorithm does not use vertex labelling, however, which is the basis of our work, and therefore we will not discuss this newer algorithm.

Ellis, Sudborough, and Turner [EST94] gave a recursive algorithm, called `COMBINE-LABELS` (Algorithm 4.1), that computes the label λ_u of a vertex u in tree T from the labels of the children of u.

Lemma 4.34 [EST94] The label of a vertex u in a tree depends only on the labels of the children of u. □

Even if our objective is to compute the vertex separation only of tree T (that is, we only need the label of the root r_T of T so that we can apply Lemma 4.19), the algorithm needs to compute the labels of all vertices in T, since in order to compute the label of r_T, it needs the labels of the children of r_T, and so on. Because of Lemma 4.34, we may make the following definition that is independent of the underlying tree T. The label λ_u of vertex u is called the *combination of the labels of the children of* u; that is, given a sequence of labels $\mu_1, \ldots, \mu_d$, label λ_u is the combination of $\mu_1, \ldots, \mu_d$ if it is the label of vertex u whose children have labels $\mu_1, \ldots, \mu_d$.

We now give a few details of algorithm COMBINE-LABELS, since in Chapter 6 we modify the algorithm slightly in order to achieve faster running time. The algorithm is an iterative one, building incrementally the label λ_u of vertex u from the sequence σ of the labels $\mu_1, \ldots, \mu_d$ of the children of u. The idea is that after the end of the kth iteration, the algorithm has computed the combination of labels in sequence $\sigma^k = \mu_1^k \ldots, \mu_d^k$, where label μ_i^k is obtained from label μ_i by deleting all elements greater than k; labels μ_i^k is defined formally below (Definition 4.5). If the last element of μ_i is greater than k, then by Lemma 4.20 all elements of μ_i are greater than k; in this case, "empty label" μ_i^k is removed from sequence σ^k. The loop terminates after the iteration in which $k = M$, where M is the maximum element of a label in sequence σ. Since Lemma 4.20 implies $\mu_i = \mu_i^M$, after it exits the loop the algorithm will have computed the combination λ_u of labels in sequence $\sigma^M = \sigma$. Before we describe more thoroughly how the algorithm works, we say more about labels $\mu_1^k \ldots, \mu_d^k$, since a similar technique for computing vertex labels will be used in the algorithm of Chapter 5 that updates the vertex labelling of the combined tree after two trees are attached at their roots.

The process of deleting from the label μ_w of a vertex w in tree T all elements greater than k to form label μ_w^k corresponds in a natural way to removing from tree T_w the subtrees corresponding to those elements. In order to discuss this in more detail, we use the notation of Definition 4.4, except that $\mu_w = \lambda_u$ in the definition; we use μ_w instead of λ_u to avoid confusion with the label computed by algorithm COMBINE-LABELS.

Consider label $\mu_w^{k_1-1}$ that is obtained from μ_w by deleting all elements greater than $k_1 - 1$. The only element of μ_w greater than $k_1 - 1$ is k_1 (Lemma 4.20); hence, $\mu_w^{k_1-1} = (k_2, \ldots, k_p)_c$. What does this deletion correspond to in the underlying tree T_w? According to Lemma 4.22, if $|\mu_w| = p > 1$, then $(k_2, \ldots, k_p)_c$ is the label of vertex w in tree $T_w\langle v_1 \rangle$. We remark that if $p = 1$, then $v_1 = w$, and therefore $T_w\langle v_1 \rangle$ is the empty tree; correspondingly, the label $\mu_w^{k_1-1}$ is undefined, since $T_w\langle v_1 \rangle$ does not contain vertex w. In order to avoid having to explicitly refer to the vertex v_1, we denote the tree $T_w\langle v_1 \rangle$ by $T_w^{k_1-1}$ (a formal definition appears below). If $k_1 - 2 \geq k_2$, then $\mu_w^{k_1-1} = \mu_w^{k_1-2}$, and correspondingly $T_w^{k_1-1} = T_w^{k_1-2}$. Assuming $|\mu_w| \geq 3$, deleting from $\mu_w^{k_1-1}$ element k_2 gives label $\mu_w^{k_2-1} = (k_3, \ldots, k_p)_c$, which by

Lemma 4.22 is the label of w in tree $T_w\langle v_1, v_2\rangle = T_w^{k_2-1}$. We can continue in this manner until all elements of μ_w have been deleted, in which case the corresponding tree is the empty tree.

Definition 4.5 [EST94] Given a vertex w in a tree T with label $\mu_w = (k_1, \ldots, k_p)_c$, we define tree T_w^t for a nonnegative integer t in the following way:

$$T_w^t = \begin{cases} T_w & \text{if } t \geq k_1; \\ T_w^{t+1} & \text{if } t < k_1 \text{ and } t+1 \neq k_i \text{ for all } i,\ 1 \leq i \leq p; \text{ and} \\ T_w^{t+1}\langle v_i\rangle & \text{if } t+1 = k_i \text{ for some } i,\ 1 \leq i \leq p; \end{cases}$$

where v_i is the vertex in T_w corresponding to element k_i of label μ_w. If tree T_w^t is not empty, the label of vertex w in T_w^t is denoted by μ_w^t.

The first case in Definition 4.5 corresponds to the base case of the recursive definition of tree T_w^t. It is also easy to see from Lemma 4.20 that if k_i is the smallest integer of label $\mu_w = (k_1, \ldots, k_p)_c$ such that $k_i > t$, tree T_w^t equals tree $T_w\langle v_1, \ldots, v_i\rangle$, where $v_1, \ldots, v_i$ are the vertices in T_w corresponding to elements $k_1, \ldots, k_i$ of μ_w, respectively. The following result is a simple consequence of Lemma 4.22 and Definition 4.5.

Lemma 4.35 [EST94] Given a tree T, the label $\mu_w = (k_1, \ldots, k_p)_c$ of a vertex w in T, and a nonnegative integer t, the following two statements hold:

1. tree T_w^t is empty if and only if $t < k_p$; and

2. if $t \geq k_p$, then $\mu_w^t = (k_i, \ldots, k_p)_c$, where k_i is the largest element of μ_w satisfying $t \geq k_i$; that is, μ_w^t is a contiguous subsequence of μ_w.

We now return to the description of algorithm `COMBINE-LABELS`. In light of Definition 4.5 and Lemma 4.35, we can rephrase the loop invariant mentioned on page 60 as follows: at the end of the kth iteration, the algorithm has computed the combination of the labels of trees $T_{u_1}^k, \ldots, T_{u_d}^k$, where $u_1, \ldots, u_d$ are the children of vertex u in tree T. However, since the algorithm computes with vertex labels, not trees, we will continue its exposition in terms of the input labels; Lemma 4.35

combined with Definition 4.5 provides us with a more intuitive way to view the computation.

Because the algorithm is iterative, we first need to set up the base case of the loop. The loop iterates from 1 to M (lines 4–14 of Algorithm 4.1 on page 63), with M defined above as the largest element of a label in sequence σ of input labels $\mu_1, \ldots, \mu_d$. The loop invariant is that at the end of the kth iteration label λ is the combination of labels in sequence $\sigma^k = \mu_1^k, \ldots, \mu_d^k$. Thus, before the loop is entered, λ is the combination of labels in σ^0. The behaviour of algorithm `COMBINE-LABELS` before entering the loop is dictated by the following two lemmas.

Lemma 4.36 [EST94] If 0 is an element of at least one of the vertex labels $\mu_1, \ldots, \mu_d$, then the combination of labels $\mu_1^0, \ldots, \mu_d^0$ is $(0)_1$. □

Lemma 4.37 [EST94] If 0 is not an element of any of the vertex labels $\mu_1, \ldots, \mu_d$, then the combination of labels $\mu_1^0, \ldots, \mu_d^0$ is $(0)_0$. □

We observe from point 2 in Lemma 4.35 that the number of labels in sequence σ^0 that have 0 as an element is the same as the number of labels in sequence σ that have 0 as an element.

In Chapter 6, we modify algorithm `COMBINE-LABELS` slightly to improve its running time. One of the modifications is in the base case of the loop; instead of starting the iteration at 1, our algorithm starts at m, where m is not necessarily 1.

We next discuss the body of the loop of algorithm `COMBINE-LABELS`. During the kth iteration, the algorithm computes the combination of labels in sequence σ^k. This combination is computed from the number of times k occurs as an element of a label in σ^k, denoted i_k, and the combination computed during the previous iteration, which is the combination of labels in sequence σ^{k-1}. As in the previous paragraph when we considered the case when $k = 0$, the number of times k occurs as an element of a label in sequence σ^k is equal to the number of times k occurs as an element of a label in sequence σ. There are four cases, depending on whether $i_k \geq 3$, $i_k = 2$, $i_k = 1$, or $i_k = 0$. In the last case, the iteration does not modify label λ, since in this case, point 2 in Lemma 4.35 implies $\sigma^k = \sigma^{k-1}$, and hence the combinations of labels in sequences σ^k and σ^{k-1} are equal. The following lemma gives the answer when $i_k \geq 3$.

Lemma 4.38 [EST94] If $k \geq 1$ is an element of at least three labels in $\mu_1, \ldots, \mu_d$, then the combination of labels $\mu_1^k, \ldots, \mu_d^k$ is $(k+1)_0$. □

The next two lemmas give label λ when $i_k = 2$.

Lemma 4.39 [EST94] If $k \geq 1$ is an element of exactly two labels in $\mu_1, \ldots, \mu_d$, and at least one of these elements is critical, then the combination of labels $\mu_1^k, \ldots, \mu_d^k$ is $(k+1)_0$. □

Lemma 4.40 [EST94] If $k \geq 1$ is an element of exactly two labels in $\mu_1, \ldots, \mu_d$, and neither of these two elements is critical, then the combination of labels $\mu_1^k, \ldots, \mu_d^k$ is $(k)_2$. □

Finally, we consider the case when $i_k = 1$. There are three subcases.

Lemma 4.41 [EST94] If $k \geq 1$ is an element of exactly one label in $\mu_1, \ldots, \mu_d$ and is critical, and if k is an element of the combination of labels $\mu_1^{k-1}, \ldots, \mu_d^{k-1}$, then the combination of labels $\mu_1^k, \ldots, \mu_d^k$ is $(k+1)_0$. □

Lemma 4.42 [EST94] If $k \geq 1$ is an element of exactly one label in $\mu_1, \ldots, \mu_d$ and is critical, and if k is not an element of the combination λ_{k-1} of labels $\mu_1^{k-1}, \ldots, \mu_d^{k-1}$, then the combination of labels $\mu_1^k, \ldots, \mu_d^k$ is label λ_{k-1} prepended with k. □

Lemma 4.43 [EST94] If $k \geq 1$ is an element of exactly one label in $\mu_1, \ldots, \mu_d$ and is noncritical, then the combination of labels $\mu_1^k, \ldots, \mu_d^k$ is $(k)_1$. □

The body of the loop of Algorithm 4.1 corresponds directly to Lemmas 4.38, 4.39, 4.40, 4.41, 4.42, and 4.43 (lines 6, 8, 9, 12, 13, and 14, respectively).

Our modification of algorithm `COMBINE-LABELS` in Chapter 6 does not modify the body of the loop at all, and therefore we can use the correctness of the loop in `COMBINE-LABELS`, stated in the next claim, to prove the correctness of our modified algorithm. Before stating the claim, we include the pseudocode for algorithm `COMBINE-LABELS`. We remark that the algorithm handles the special case when $d = 0$; that is, it computes the combination of zero labels. This corresponds to the case when we want to compute the label of a leaf.

Algorithm 4.1 [EST94]
Input: a sequence σ of vertex labels $\mu_1, \ldots, \mu_d$.
Output: the combination λ of the labels in σ.

```
COMBINE-LABELS($\mu_1, \ldots, \mu_d$)
1.   if 0 is an element of at least one label in $\sigma$ then $\lambda \leftarrow (1)_0$
2.   else $\lambda \leftarrow (0)_0$
3.   $M \leftarrow$ the maximum element of a label in $\sigma$ if $d > 0$, and 0 if $d = 0$
4.   for $k$ from 1 to $M$ do
5.       $i_k \leftarrow$ the number of labels in $\sigma$ that contain $k$
6.       if $i_k \geq 3$ then $\lambda \leftarrow (k+1)_0$
7.       else if $i_k = 2$ then
8.           if $k$ is a critical element of at least one label in $\sigma$ then
                 $\lambda \leftarrow (k+1)_0$
9.           else $\lambda \leftarrow (k)_2$
10.      else if $i_k = 1$ then
11.          if $k$ is a critical element of the label in $\sigma$ that contains $k$ then
12.              if $\lambda(1) = k$ then $\lambda \leftarrow (k+1)_0$
13.              else $\lambda \leftarrow$ label $\lambda$ prepended with $k$
14.          else $\lambda \leftarrow (k)_1$
```

Claim 4.44 [EST94] If λ is the combination of labels $\mu_1^{j-1}, \ldots, \mu_d^{j-1}$ just before the iteration in which $k = j$ in the loop on lines 4–14 of Algorithm 4.1, then λ is the combination of labels $\mu_1^j, \ldots, \mu_d^j$ just after the iteration. □

The following lemma states the correctness and running time of the algorithm.

Lemma 4.45 [EST94] If $\sigma = \mu_1, \ldots, \mu_d$ is a sequence of d vertex labels, where $d \geq 0$, then on input $(\mu_1, \ldots, \mu_d)$ Algorithm 4.1 correctly computes the combination of the labels in σ and runs in time $\Theta(dM)$, where M is the largest element of a label in σ. □

Algorithm COMBINE-LABELS can be used to compute the vertex labelling, and therefore the vertex separation (Lemma 4.19), of tree T by recursively computing the labels of the children of the root of T and then computing the combination of these labels. The vertex labelling of tree T can in turn be used to find an optimal layout of T. The next two theorems state upper bounds on the time required to compute the vertex labelling and an optimal layout of a tree.

Theorem 4.46 [EST94] Given a rooted tree T, the vertex labelling of T can be computed in $O(|T|\mathrm{VS}(T)) = O(|T| \lg |T|)$ time using Algorithm 4.1. $\square$

Theorem 4.47 [EST94] Given a rooted tree T, an optimal layout of T with respect to vertex separation can be computed in $O(|T|\mathrm{VS}(T)) = O(|T| \lg |T|)$ time from the vertex labelling of tree T. $\square$

In the upcoming chapters, we give algorithms that use the vertex labelling of tree T to update the vertex separation of T after a tree is attached to T or a subtree is removed from T.

Chapter 5

Attaching Trees at Their Roots

In some cases, we construct a tree incrementally vertex-by-vertex, or subtree-by-subtree, and we need to maintain the correctness of the vertex labelling of such a dynamically growing tree, given that the vertex labelling of each subtree has been computed. In this chapter, we give an algorithm that updates the vertex labelling of a tree after another tree is attached to it via the edge that connects the two roots. The more general case when a tree is attached by its root to an arbitrary vertex of another tree will be considered in Chapter 7. Updating the vertex labelling of a tree after removing a subtree will be discussed in Chapters 6 and 7.

We first explain in general why it is useful to update the vertex labelling of a tree T after another tree S is attached to T. Assume the vertex labellings Λ_T and Λ_S of T and S have been computed. The problem we wish to solve is that of computing the vertex labelling Λ of the composite tree $T \vdash_a S$ from Λ_T and Λ_S, where a is the vertex of attachment in T. We will see in this chapter that the vertex separation of $T \vdash_a S$ cannot be computed from the vertex separations of trees T and S alone, but it can be computed from labellings Λ_T and Λ_S much faster than by recomputing Λ. Hence, if we maintain the correctness of the vertex labelling of tree T as other trees are attached to or removed from it, we maintain the correct value of the vertex separation of the tree. As mentioned earlier, in this chapter we only consider the case when a is one of the two roots r_T and r_S of trees T and S. If $a = r_T$, the composite tree is $T \vdash_{r_T} S = T \vdash S$, and if $a = r_S$, the composite tree is $S \vdash_{r_S} T = S \vdash T$. Without loss of generality, we assume the new tree is $T \vdash S$.

The important observation, proved below, is that λ_{r_T} is the only label of a vertex in $T \vdash S$ that can change after trees T and S are attached at their roots and the root r_T of T is made the root of the new tree.

Lemma 5.1 Given trees T and S with roots r_T and r_S, respectively, if u is a vertex in T such that $u \neq r_T$, then $\lambda_{u,T} = \lambda_{u,T\vdash S}$. If u is a vertex in S, then $\lambda_{u,S} = \lambda_{u,T\vdash S}$.

Proof. The lemma follows directly from Definition 4.4, the fact that vertex r_T is the only vertex u in tree T for which $(T \vdash S)_u \neq T_u$, and the observation that all vertices u in tree S satisfy $(T \vdash S)_u = S_u$. $\square$

Lemma 5.1 implies that in order to compute the vertex labelling Λ of tree $T \vdash S$ from labellings Λ_T and Λ_S, we only need to update the label λ_{r_T}. Algorithm 4.1 can compute the label λ of vertex r_T in tree $T \vdash S$ from the labels of the d children of r_T in $T \vdash S$. This can take time $\Omega(|T \vdash S|)$ using Algorithm 4.1, because of Lemma 4.45 and the observation that d can be $\Omega(|T \vdash S|)$. Our algorithm computes λ only from labels λ_T and λ_S, achieving $O(\lg |T \vdash S|)$ running time. Labels λ, λ_T, and λ_S are the labels of trees $T \vdash S$, T, and S, since they are the labels of the roots of $T \vdash S$, T, and S, respectively.

5.1 Description of the Algorithm

In computing the label λ of tree $T \vdash S$, our algorithm, called `ADD-LABEL`, employs the technique used in Algorithm 4.1 (`COMBINE-LABELS`). It builds λ incrementally, from the smallest element to the largest. We denote by λ_k the label of the tree $T^k \vdash S^k$, where k is a nonnegative integer. We remark that if T^k is the empty tree, then $T^k \vdash S^k$ is empty, and therefore λ_k is undefined. We denote by m the maximum of $\{1, \lambda_T(|\lambda_T|)\}$, where $\lambda_T(|\lambda_T|)$ is the last element of label λ_T, and by M the maximum vertex separation of trees T and S. Lemma 4.19 implies $\mathrm{VS}(T) = \lambda_{r_T,T}(1) = \lambda_T(1)$ and $\mathrm{VS}(S) = \lambda_{r_S,S}(1) = \lambda_S(1)$, and hence $M = \max\{\lambda_T(1), \lambda_S(1)\}$.

Algorithm `ADD-LABEL` iteratively computes a sequence of the labels $\lambda_0, \lambda_m, \lambda_{m+1}, \ldots, \lambda_M$ of trees $T^0 \vdash S^0, T^m \vdash S^m, T^{m+1} \vdash S^{m+1}, \ldots, T^M \vdash S^M$, respectively. Since $m \geq \lambda_T(|\lambda_T|)$, point 1 in Lemma 4.35 implies tree T^k, and therefore tree $T^k \vdash S^k$, is nonempty for all $k \geq m$, and hence all labels in the sequence are defined, except possibly λ_0, which is undefined if and only if $\lambda_T(|\lambda_T|) > 0$, which again follows from point 1 in Lemma 4.35. The fact that $M \geq \lambda_T(1)$ and $M \geq \lambda_S(1)$ implies $T^M = T$ and $S^M = S$ (the first case in Definition 4.5), and therefore the last label in the sequence is λ, the label of tree $T \vdash S$.

During each iteration in the algorithm, λ_k is computed from λ_{k-1} and labels λ_T^k and λ_S^k. We recall that λ_T^k and λ_S^k are the labels of trees T^k and S^k, respectively, and point 2 in Lemma 4.35 implies they are contiguous subsequences of λ_T and λ_S. There are three cases to consider, depending on whether k is an element of both λ_T^k and λ_S^k, it is an element of exactly one of the labels, or it is an element of neither label. In the first case, the subroutine `ADD-LABEL-EQUAL` is called, and in the second case, subroutine `ADD-LABEL-UNEQUAL` is invoked to compute λ_k. If k is an element of neither λ_T^k nor λ_S^k, then it follows from point 2 in Lemma 4.35 that k is an element of neither λ_T nor λ_S. Hence, $T^k = T^{k-1}$ and $S^k = S^{k-1}$ (the second case in Definition 4.5), which implies $T^k \vdash S^k = T^{k-1} \vdash S^{k-1}$. Therefore, in this case, $\lambda_k = \lambda_{k-1}$ holds, and the iteration does nothing. We give the pseudocode for algorithms `ADD-LABEL-EQUAL` and `ADD-LABEL-UNEQUAL` before giving the pseudocode for `ADD-LABEL`. The correctness and running time of all three algorithms are proved afterward.

Subroutine `ADD-LABEL-EQUAL` is called on arguments λ_T^k and λ_S^k from algorithm `ADD-LABEL`, and assuming that k is an element of both labels λ_T^k and λ_S^k, it returns the label λ_k. By Lemmas 4.20 and 4.35, the largest element of both λ_T^k and λ_S^k is k, and hence the assumption can be restated as $\lambda_T^k(1) = k$ and $\lambda_S^k(1) = k$. We briefly discuss what algorithm `ADD-LABEL-EQUAL` does; the correctness proof will appear later (Lemma 5.7). The algorithm has two cases, depending on whether k is a critical element of either λ_T^k or λ_S^k, or it is a critical element of neither label. In the former case, we will show later (Lemma 5.5) that the vertex separation of tree $T^k \vdash S^k$ is $k+1$, there is no critical vertex in $T^k \vdash S^k$, and the criticality of the root of $T^k \vdash S^k$ is 0. Thus, the label λ_k is $(k+1)_0$. In the latter case, we will prove (Lemma 5.6) that attaching trees T^k and S^k at their roots does not increase the vertex separation, the criticality of the root of T^k increases by one, and there is no critical vertex in $T^k \vdash S^k$, except possibly the root of T^k. Therefore,

$\lambda_k = (k)_{crit\left(\lambda_T^k\right)+1}$.

Algorithm 5.1
Input: the vertex labels λ_T^k and λ_S^k of trees T and S, respectively, such that $\lambda_T^k(1) = \lambda_S^k(1) = k$.
Output: the vertex label λ_k of tree $T^k \vdash S^k$.

```
ADD-LABEL-EQUAL $\left(\lambda_T^k, \lambda_S^k\right)$
1.      if $k$ is a critical element of either $\lambda_T$ or $\lambda_S$ then $\lambda_k \leftarrow (k+1)_0$
2.      else $\lambda_k \leftarrow (k)_{crit\left(\lambda_T^k\right)+1}$
```

Subroutine ADD-LABEL-UNEQUAL is called on arguments λ_T^k, λ_S^k, and λ_{k-1}, where λ_{k-1} is the label computed during the previous iteration in algorithm ADD-LABEL; that is, it is the label of tree $T^{k-1} \vdash S^{k-1}$. The precondition of algorithm ADD-LABEL-UNEQUAL is that k is an element of exactly one of the labels λ_T^k and λ_S^k. By point 2 in Lemma 4.35, the largest elements of λ_T^k and λ_S^k are at most k, and hence the precondition is equivalent to $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$ and $\lambda_T^k(1) \neq \lambda_S^k(1)$.

We now discuss each of the four cases covered in the algorithm. The first case occurs when k is the only element of label λ_T^k. We will show in Lemma 5.8 that the label of tree T^k does not change as tree S^k is attached to it, and hence $\lambda_k = \lambda_T^k = (k)_{crit\left(\lambda_T^k\right)}$. If the first case does not apply, and if k is the only element of λ_S^k and is noncritical, then we will show (Lemma 5.9) that the vertex separation of tree $T^k \vdash S^k$ is k, there is no critical vertex in $T^k \vdash S^k$, and the criticality of the root of $T^k \vdash S^k$ is 1. Therefore, $\lambda_k = (k)_1$. Argument λ_{k-1} is not used in either of the two cases. If neither the first nor second case applies, then k is a critical element of either λ_T^k or λ_S^k (this will be shown in Lemma 5.12). We will also prove that $T^{k-1} \vdash S^{k-1}$ is nonempty, thus guaranteeing label λ_{k-1} is defined. If $\lambda_{k-1}(1) = \mathrm{VS}\left(T^{k-1} \vdash S^{k-1}\right) < k$, we will show that λ_k is label λ_{k-1} prepended with k (Lemma 5.10); otherwise, $\lambda_k = (k+1)_0$ (Lemma 5.11).

Algorithm 5.2

Input: the vertex labels λ_T^k and λ_S^k of trees T^k and S^k, respectively, such that $\lambda_T^k(1) \neq \lambda_S^k(1)$, and the label λ_{k-1} of $T^{k-1} \vdash S^{k-1}$, where $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$. If $T^{k-1} \vdash S^{k-1}$ is empty, then argument λ_{k-1} is not used (it is not defined in this case).

Output: the vertex label λ_k of tree $T^k \vdash S^k$.

```
ADD-LABEL-UNEQUAL $\left(\lambda_T^k, \lambda_S^k, \lambda_{k-1}\right)$
1.    if $k$ is the only element of $\lambda_T^k$ then $\lambda_k \leftarrow (k)_{crit\left(\lambda_T^k\right)}$
2.    else if $k$ is the only element of $\lambda_S^k$ and is noncritical then $\lambda_k \leftarrow (k)_1$
3.    else if $\lambda_{k-1}(1) < k$ then $\lambda_k \leftarrow \lambda_{k-1}$ prepended with $k$
4.    else $\lambda_k \leftarrow (k+1)_0$
```

Finally, we present the pseudocode for the main iterative algorithm, which calls the subroutines just described. Before the algorithm enters the main loop, it computes the label of tree $T^0 \vdash S^0$. The label remains undefined unless the smallest element of λ_T is 0, in which case T^0 consists of the single vertex r_T. Tree S^0 is also either empty or consists of the single vertex r_S. Hence, tree $T^0 \vdash S^0$ is either the empty tree, the single vertex r_T, or the path r_T, r_S. The base case is handled by the algorithm before entering the main loop. Values m and M are used by the algorithm in the same way as they are defined on page 68.

Algorithm 5.3

Input: the vertex labels λ_T and λ_S of trees T and S, respectively.

Output: the vertex label λ of tree $T \vdash S$.

```
ADD-LABEL($\lambda_T, \lambda_S$)
1.    if $\lambda_T(|\lambda_T|) = 0$ then
2.        if $\lambda_S(|\lambda_S|) > 0$ then $\lambda \leftarrow (0)_0$
3.        else $\lambda \leftarrow (1)_0$
4.    $m \leftarrow \max\{1, \lambda_T(|\lambda_T|)\}$; $M \leftarrow \max\{\lambda_T(1), \lambda_S(1)\}$
5.    for $k$ from $m$ to $M$ do
6.        if $k$ is an element of both $\lambda_T$ and $\lambda_S$ then
7.            $\lambda \leftarrow$ ADD-LABEL-EQUAL $\left(\lambda_T^k, \lambda_S^k\right)$
8.        else if $k$ is an element of exactly one of $\lambda_T$ and $\lambda_S$ then
9.            $\lambda \leftarrow$ ADD-LABEL-UNEQUAL $\left(\lambda_T^k, \lambda_S^k, \lambda\right)$
```

5.2 Correctness and Running Time of the Algorithm

Before proving the correctness of algorithms `ADD-LABEL-EQUAL`, `ADD-LABEL-UNEQUAL`, and finally `ADD-LABEL`, we prove three simple claims that are used several times in the correctness proofs. The first claim is used to infer the label of tree $T^k \vdash S^k$ if its vertex separation is greater than k.

Claim 5.2 Given trees T^k and S^k such that $\mathrm{VS}\left(T^k \vdash S^k\right) > k$, the label of tree $T^k \vdash S^k$ is $(k+1)_0$.

Proof. We first show that $\mathrm{VS}\left(T^k \vdash S^k\right) = k+1$ and that the criticality of the root r_T in tree $T^k \vdash S^k$ is 0. Next, we prove that there is no critical vertex in $T^k \vdash S^k$. This implies by Lemma 4.21 that the label of the root r_T in $T^k \vdash S^k$ is $(k+1)_0$. Before we proceed with the rest of the proof, we observe the following: point 2 in Lemma 4.35 implies the largest elements of the labels of trees T^k and S^k are at most k, and hence Lemma 4.19 yields $\mathrm{VS}\left(T^k\right) \le k$ and $\mathrm{VS}\left(S^k\right) \le k$.

First, we show that $\mathrm{VS}\left(T^k \vdash S^k\right) \le k+1$, which means $\mathrm{VS}\left(T^k \vdash S^k\right) = k+1$, because of the fact $\mathrm{VS}\left(T^k \vdash S^k\right) > k$. The proof of $\mathrm{VS}\left(T^k \vdash S^k\right) \le k+1$ involves showing that root r_T is a $(k+1)$-backbone of tree $T^k \vdash S^k$. Since every branch of r_T in $T^k \vdash S^k$ is either a branch of r_T in T^k or it is the tree S^k, $\left(T^k \vdash S^k\right)[r_T] = T^k[r_T] \cup \left\{S^k\right\}$. By Lemma 2.1 and the fact $\mathrm{VS}\left(T^k\right) \le k$, every subtree in $T^k[r_T]$ has vertex separation at most k. And because $\mathrm{VS}\left(S^k\right) \le k$, every subtree in $\left(T^k \vdash S^k\right)[r_T]$ has vertex separation at most k. This implies r_T is a $(k+1)$-backbone of $T^k \vdash S^k$, and hence Lemma 4.3 implies $\mathrm{VS}\left(T^k \vdash S^k\right) \le k+1$. Therefore, $\mathrm{VS}\left(T^k \vdash S^k\right) = k+1$. Since the set of subtrees yielded by root r_T is $\left(T^k \vdash S^k\right)[r_T]$, no subtree yielded by r_T has vertex separation greater than k. Thus, the criticality of r_T in $T^k \vdash S^k$ is 0.

Next, we show by a straightforward application of Lemma 2.1 that there is no critical vertex in tree $T^k \vdash S^k$. Given an arbitrary vertex u in $T^k \vdash S^k$ such that $u \ne r_T$, Lemma 5.1 implies $\left(T^k \vdash S^k\right)_u = \left(X^k\right)_u$, where $X^k = T^k$ if u is in T^k and $X^k = S^k$ if u is in S^k. It follows from the facts $\mathrm{VS}\left(T^k\right) \le k$ and $\mathrm{VS}\left(S^k\right) \le k$ that $\mathrm{VS}\left(X^k\right) \le k$, and Lemma 2.1 implies the vertex separation of $\left(T^k \vdash S^k\right)_u$ is at most k. We conclude from Lemma 2.1 that tree $\left(T^k \vdash S^k\right)_u$ has no subtrees with vertex separation greater than k, and therefore u cannot be critical. Hence, there is no critical vertex in tree $T^k \vdash S^k$. □

The second claim places restrictions on which vertices in tree $T^k \vdash S^k$ can be critical.

Claim 5.3 Given trees T^k and S^k such that $\mathrm{VS}\left(T^k \vdash S^k\right) = k$, if tree $R = T^k \vdash S^k$ has a critical vertex $\mathit{inf}_R \neq r_T$, then either $\mathrm{VS}\left(T^k\right) = k$ and inf_R is critical in T^k, or $\mathrm{VS}\left(S^k\right) = k$ and inf_R is critical in S^k.

Proof. Suppose tree $T^k \vdash S^k$ has a critical vertex $\mathit{inf}_R \neq r_T$. Lemma 5.1 implies $\left(T^k \vdash S^k\right)_{\mathit{inf}_R} = \left(X^k\right)_{\mathit{inf}_R}$, where $X^k = T^k$ if inf_R is in T^k and $X^k = S^k$ if inf_R is in S^k. It follows from Lemma 2.1 that $\mathrm{VS}\left(\left(T^k \vdash S^k\right)_{\mathit{inf}_R}\right) = k$, and hence $\mathrm{VS}\left(\left(X^k\right)_{\mathit{inf}_R}\right) = k$. Since $\left(X^k\right)_{\mathit{inf}_R}$ is a subtree of X^k and X^k is a subtree of $T^k \vdash S^k$, Lemma 2.1 implies $\mathrm{VS}\left(X^k\right) = k$. We conclude that vertex inf_R is critical in tree X^k, because the subtrees yielded by inf_R in X^k are the same as the subtrees yielded by inf_R in $T^k \vdash S^k$. □

The third claim essentially shows that removing a subtree rooted at a vertex u from either tree T^k or S^k such that $u \neq r_T$, and then attaching the trees at their roots gives the same result as attaching trees T^k and S^k first, and then removing the subtree rooted at u from $T^k \vdash S^k$. The actual statement of this result is more technical and tailored for its use later.

Claim 5.4 Given trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that k is an element of exactly one label and there is an element of λ_T^k smaller than k, the trees $T^{k-1} \vdash S^{k-1}$ and $\left(T^k \vdash S^k\right)\langle v_k\rangle$ are identical, where v_k is the vertex in either T^k or S^k corresponding to element k.

Proof. We prove the claim in two stages. We denote by X^k the tree T^k or S^k such that k is an element of label λ_X^k, and by Y^k the other tree; that is, $Y^k = T^k$ if $X^k = S^k$ and $Y^k = S^k$ if $X^k = T^k$. First, we show that $X^{k-1} = X^k\langle v_k\rangle$ and $Y^{k-1} = Y^k$. Since there is an element of label λ_T^k smaller than k, point 1 in Lemma

4.35 implies tree T^{k-1} is nonempty. We therefore infer the following: if $X^k = T^k$, then $T^{k-1} \vdash S^{k-1} = T^k\langle v_k\rangle \vdash S^k$, and if $X^k = S^k$, then $T^{k-1} \vdash S^{k-1} = T^k \vdash S^k\langle v_k\rangle$. Second, we prove that $T^k\langle v_k\rangle \vdash S^k = \left(T^k \vdash S^k\right)\langle v_k\rangle$ if $X^k = T^k$ and $T^k \vdash S^k\langle v_k\rangle = \left(T^k \vdash S^k\right)\langle v_k\rangle$ if $X^k = S^k$, which shows that $T^{k-1} \vdash S^{k-1} = \left(T^k \vdash S^k\right)\langle v_k\rangle$.

By applying the recursive definition of trees X^{k-1} and Y^{k-1} (Definition 4.5), we first show that $X^{k-1} = X^k\langle v_k\rangle$ and $Y^{k-1} = Y^k$. Since k is an element of label λ^k_X, tree X^{k-1} is obtained from tree X^k by removing the subtree rooted at v_k. Thus, $X^{k-1} = X^k\langle v_k\rangle$. Because k is an element of exactly one of the labels λ^k_T and λ^k_S, it follows that k is not an element of λ^k_Y. Hence, $Y^{k-1} = Y^k$.

Second, we show that $T^k\langle v_k\rangle \vdash S^k = \left(T^k \vdash S^k\right)\langle v_k\rangle$ if $X^k = T^k$ and $T^k \vdash S^k\langle v_k\rangle = \left(T^k \vdash S^k\right)\langle v_k\rangle$ if $X^k = S^k$. We prove this by showing that $v_k \neq r_T$ and then applying Lemma 5.1. We first consider the case when $X^k = T^k$. Since there is an element of label λ^k_T smaller than k, it follows from Lemma 4.20 that element k is not the last element of λ^k_T, and hence the vertex v_k corresponding to k is not the root r_T (Lemma 4.27). Next, we consider the case when $X^k = S^k$. Then $v_k \neq r_T$, because r_T is not in tree S^k. Therefore, in both cases, $v_k \neq r_T$ holds. Lemma 5.1 now implies $\left(X^k\right)_{v_k} = \left(T^k \vdash S^k\right)_{v_k}$. Therefore, if $X^k = T^k$, tree $\left(T^k \vdash S^k\right)\langle v_k\rangle$ is obtained from tree $T^k \vdash S^k$ by removing subtree $\left(T^k \vdash S^k\right)_{v_k} = \left(T^k\right)_{v_k}$, and thus $\left(T^k \vdash S^k\right)\langle v_k\rangle = T^k\langle v_k\rangle \vdash S^k$. Similarly, if $X^k = S^k$, then $\left(T^k \vdash S^k\right)\langle v_k\rangle$ is obtained from $T^k \vdash S^k$ by removing subtree $\left(T^k \vdash S^k\right)_{v_k} = \left(S^k\right)_{v_k}$, implying $\left(T^k \vdash S^k\right)\langle v_k\rangle = T^k \vdash S^k\langle v_k\rangle$. □

We now show the correctness of algorithm `ADD-LABEL-EQUAL` (Algorithm 5.1) by first proving two lemmas directly corresponding to the two cases on lines 1 and 2 and then combining them. Both lemmas have as one of their conditions $\lambda^k_T(1) = \lambda^k_S(1) = k$, where λ^k_T and λ^k_S are the labels of trees T^k and S^k, respectively; this is a precondition of the algorithm. In the first lemma, we consider the case when k is a critical element of either λ^k_T or λ^k_S (line 1 of the algorithm).

Lemma 5.5 Suppose we have trees T^k and S^k with labels λ^k_T and λ^k_S, respectively, such that $\lambda^k_T(1) = \lambda^k_S(1) = k$. If k is a critical element of either λ^k_T or λ^k_S (or both), then the label λ_k of tree $T^k \vdash S^k$ is $(k+1)_0$.

Proof. We show that $\mathrm{VS}\left(T^k \vdash S^k\right) > k$. Claim 5.2 then implies $\lambda_k = (k+1)_0$. We prove $\mathrm{VS}\left(T^k \vdash S^k\right) > k$ by showing there is a vertex in tree $T^k \vdash S^k$ that has three branches with vertex separation at least k; Lemma 4.4 then implies $\mathrm{VS}\left(T^k \vdash S^k\right) > k$. Since k is a critical element of either λ_T^k or λ_S^k, there is a critical vertex in T^k or S^k. Consider the critical vertex inf_{X^k} in tree X^k, where $X^k = T^k$ or $X^k = S^k$, and the critical children inf_1 and inf_2 of inf_{X^k}. Vertex inf_{X^k} has three branches in $T^k \vdash S^k$ that have vertex separation at least k: $\left(X^k\right)_{\mathit{inf}_1}$, $\left(X^k\right)_{\mathit{inf}_2}$, and Y, where Y is defined as follows. If inf_{X^k} is the root r_T of T^k, then $Y = S^k$, since tree S^k is a branch of r_T in $T^k \vdash S^k$ and $\mathrm{VS}\left(S^k\right) = k$ (Figure 5.1). If $\mathit{inf}_{X^k} \neq r_T$, then

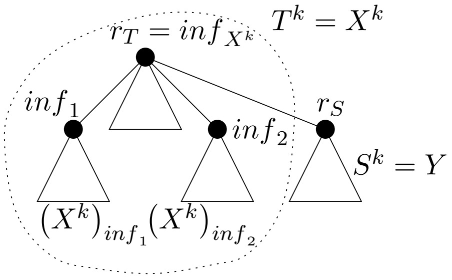

FIGURE 5.1: The proof of Lemma 5.5 when $\mathit{inf}_{X^k} = r_T$.

$Y = \left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle$, since the branch $\left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle$ of inf_{X^k} in $T^k \vdash S^k$ contains S^k or T^k as a subtree, and hence by Lemma 2.1 the vertex separation of $\left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle$ is at least k; if $X^k = T^k$, then $\left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle$ contains tree S^k as a subtree (Figure 5.2), and if $X^k = S^k$, then $\left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle$ contains tree T^k as a subtree (Figure 5.3). Thus, vertex inf_{X^k} has three branches in tree $T^k \vdash S^k$ with vertex separation at least k. □

In the second lemma, we consider the case when the condition of Lemma 5.5 does not hold; that is, when k is a noncritical element of both labels λ_T^k and λ_S^k. This case corresponds to line 2 of algorithm `ADD-LABEL-EQUAL`.

Lemma 5.6 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that $\lambda_T^k(1) = \lambda_S^k(1) = k$. If k is a noncritical element of both λ_T^k and λ_S^k, then the label of tree $T^k \vdash S^k$ is $(k)_{crit\left(\lambda_T^k\right)+1}$.

Proof. The first step of our proof is a construction of a k-backbone of tree $T^k \vdash S^k$. Lemma 4.3 then implies the vertex separation of $T^k \vdash S^k$ is at most k. The fact

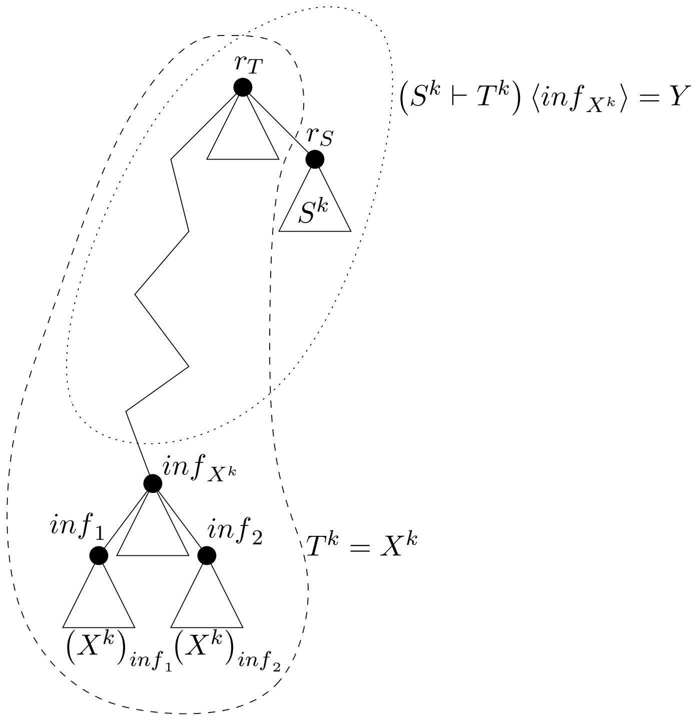
r_T
r_S
S^k
$\left(S^k \vdash T^k\right)\langle inf_{X^k}\rangle = Y$
inf_{X^k}
inf_1
inf_2
$T^k = X^k$
$\left(X^k\right)_{inf_1}$
$\left(X^k\right)_{inf_2}$

FIGURE 5.2: The proof of Lemma 5.5 when $inf_{X^k} \neq r_T$ and $X^k = T^k$.

$\lambda_T^k(1) = \lambda_S^k(1) = k$ and Lemma 4.19 imply VS $\left(T^k\right) = $ VS $\left(S^k\right) = k$. Therefore, Lemma 2.1 yields VS $\left(T^k \vdash S^k\right) = k$, because T^k and S^k are subtrees of $T^k \vdash S^k$. We next prove there is no critical vertex inf_R in tree $R = T^k \vdash S^k$ such that $inf_R \neq r_T$, and then finally conclude that the label of $T^k \vdash S^k$ is $(k)_{crit\left(\lambda_T^k\right)+1}$.

First, we construct a k-backbone P of tree $T^k \vdash S^k$ from the canonical backbones B_{T^k} and B_{S^k} of trees T^k and S^k. Since k is the first element of both labels λ_T^k and λ_S^k and is noncritical, by Lemma 4.29

$$\left|\lambda_T^k\right| = \left|\lambda_S^k\right| = 1. \tag{5.1}$$

Thus, both λ_T^k and λ_S^k are noncritical labels, because the last element of each label is noncritical. Lemma 4.32 therefore implies the root r_T of T^k is an endpoint of B_{T^k} and the root r_S of S^k is an endpoint of B_{S^k} (Figure 5.4). Since the concept of the canonical backbone is independent of its orientation, we may assume without loss

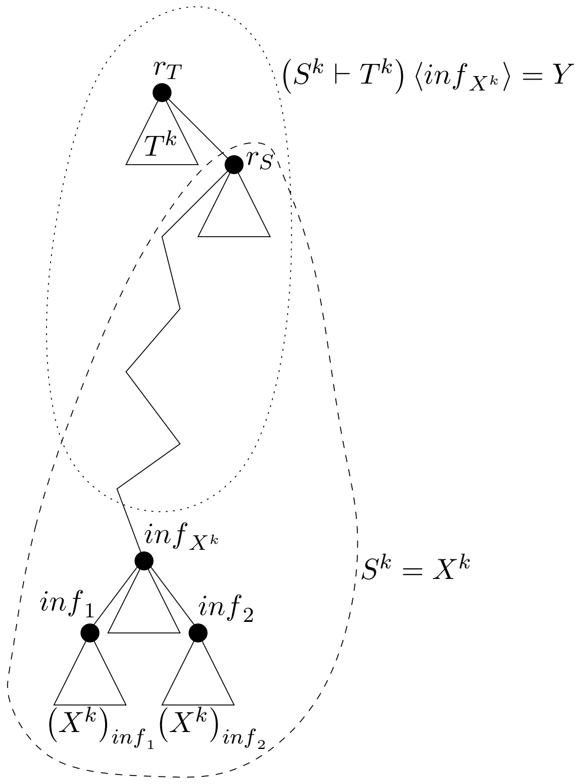
r_T
$\left(S^k \vdash T^k\right)\langle inf_{X^k}\rangle = Y$
T^k
r_S
inf_{X^k}
inf_1
inf_2
$S^k = X^k$
$\left(X^k\right)_{inf_1}$
$\left(X^k\right)_{inf_2}$

FIGURE 5.3: The proof of Lemma 5.5 when $inf_{X^k} \neq r_T$ and $X^k = S^k$.

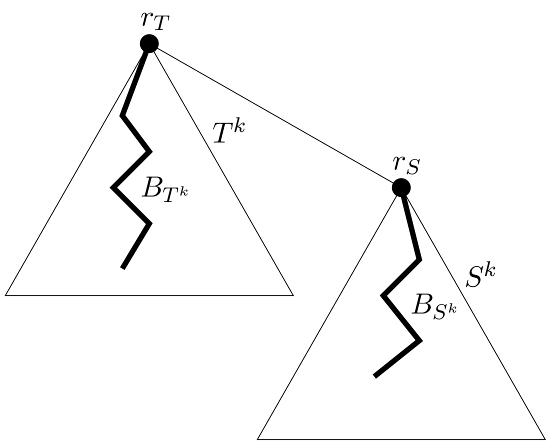

FIGURE 5.4: The proof of Lemma 5.6.

of generality that r_T is the right endpoint of B_{T^k} and r_S is the left endpoint of B_{S^k}. Because $r_T r_S$ is an edge in tree $T^k \vdash S^k$, we may consider the path $P = B_{T^k} + B_{S^k}$ in $T^k \vdash S^k$. All branches of both B_{T^k} in T^k and B_{S^k} in S^k have vertex separation less than k, and therefore all branches of P in $T^k \vdash S^k$ have vertex separation less than k. Hence, P is a k-backbone of $T^k \vdash S^k$.

We next show that if there is a critical vertex inf_R in tree $R = T^k \vdash S^k$, then $\mathit{inf}_R = r_T$. Suppose for sake of contradiction that tree $T^k \vdash S^k$ has a critical vertex $\mathit{inf}_R \neq r_T$. Then Claim 5.3 implies VS $\left(T^k\right) = k$ and inf_R is a critical vertex in T^k, or VS $\left(S^k\right) = k$ and inf_R is a critical vertex in S^k. We denote by X^k the tree T^k or S^k, depending on whether vertex inf_R is in T^k or S^k, respectively. Since VS $\left(T^k\right) =$ VS $\left(S^k\right) = k$, VS $\left(X^k\right) = k$. But if inf_R is critical in X^k, Lemma 4.24 is contradicted, because of Equation 5.1 and the fact, shown right after the equation, that both labels λ_T^k and λ_S^k are noncritical. We conclude that if tree $T^k \vdash S^k$ has critical vertex inf_R, then $\mathit{inf}_R = r_T$.

We now derive the label λ_k of tree $T^k \vdash S^k$ and then show its criticality, concluding that $\lambda_k = (k)_{\mathit{crit}\left(\lambda_T^k\right)+1}$. Since we just proved that if $T^k \vdash S^k$ has a critical vertex inf_R, then $\mathit{inf}_R = r_T$, we infer from Lemma 4.21 and the fact VS $\left(T^k \vdash S^k\right) = k$, shown earlier, that the label λ_k of $T^k \vdash S^k$ is $(k)_{\mathit{crit}\left(r_T, T^k \vdash S^k\right)}$.

We now prove that the criticality of root r_T in tree $T^k \vdash S^k$ is $\mathit{crit}\left(\lambda_T^k\right) + 1$. The criticality of r_T in $T^k \vdash S^k$ is equal to the number of subtrees yielded by r_T in $T^k \vdash S^k$ that have vertex separation k. The number of such subtrees in $T^k \vdash S^k$ is one more than the number of such subtrees in tree T^k, since tree S^k, which has vertex separation k, is attached to r_T in constructing the tree $T^k \vdash S^k$; that is, $\mathit{crit}\left(r_T, T^k \vdash S^k\right) = \mathit{crit}\left(r_T, T^k\right) + 1$. Since Equation 5.1 yields $\left|\lambda_T^k\right| = 1$, Lemma 4.27 implies the vertex corresponding to the only element k of λ_T^k is r_T. Thus, the criticality of r_T in tree T^k equals the criticality of λ_T^k; that is, $\mathit{crit}\left(r_T, T^k\right) = \mathit{crit}\left(\lambda_T^k\right)$. We conclude that the criticality of r_T in $T^k \vdash S^k$ is $\mathit{crit}\left(\lambda_T^k\right) + 1$. □

The correctness of algorithm `ADD-LABEL-EQUAL` (Algorithm 5.1) follows easily from Lemmas 5.5 and 5.6.

Lemma 5.7 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively. If $\lambda_T^k(1) = \lambda_S^k(1) = k$, then on input $\left(\lambda_T^k, \lambda_S^k\right)$ Algorithm 5.1 correctly computes the label λ_k of tree $T^k \vdash S^k$.

Proof. The proof is a simple case analysis. Condition of Lemma 5.5 is the same as the condition on line 1 of the algorithm: $k = \lambda_T^k(1) = \lambda_S^k(1)$ is a critical element of either λ_T^k or λ_S^k. In this case, the algorithm returns $\lambda_k = (k+1)_0$ as the label of tree $T^k \vdash S^k$, which is correct (Lemma 5.5). If k is a noncritical element of both λ_T^k and λ_S^k, then the algorithm returns $\lambda_k = (k)_{crit\left(\lambda_T^k\right)+1}$ on line 2, which is correct (Lemma 5.6). □

We show in a similar fashion that algorithm `ADD-LABEL-UNEQUAL` (Algorithm 5.2) is correct; we prove four lemmas corresponding to lines 1, 2, 3, and 4 of the algorithm, and then combine them. The four lemmas have the condition $\lambda_T^k(1) \neq \lambda_S^k(1)$ in common. In each of the lemmas, k is the first element of exactly one of the labels λ_T^k and λ_S^k, where $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$, and the first element of the other label is smaller than k. In the first lemma, we consider the case when k is the only element of label λ_T^k (line 1 of algorithm `ADD-LABEL-UNEQUAL`).

Lemma 5.8 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that $\lambda_T^k(1) \neq \lambda_S^k(1)$ and $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$. If k is the only element of λ_T^k, then the label of tree $T^k \vdash S^k$ is $(k)_{crit\left(\lambda_T^k\right)} = \lambda_T^k$.

Proof. The proof involves three stages. We first describe how to tie the results of the three stages together to reach the conclusion $\lambda_k = (k)_{crit\left(\lambda_T^k\right)}$, and then prove each of the three results in turn. First, we show that the root r_T of tree T^k is in B_{T^k}, implying tree S^k is a branch in tree $T^k \vdash S^k$ of the canonical backbone B_{T^k} of tree T^k. Lemma 4.3 then implies VS $\left(T^k \vdash S^k\right) \leq k$, since every branch of B_{T^k} in $T^k \vdash S^k$ is either a branch of B_{T^k} in T^k, and hence has vertex separation less than k, or it is tree S^k; but by Lemma 4.19 VS $\left(S^k\right) < k$. Therefore, Lemma 2.1 implies VS $\left(T^k \vdash S^k\right) = k$, since T^k is a subtree of $T^k \vdash S^k$ and VS $\left(T^k\right) = k$. Second, we show that there is no critical vertex inf_R in tree $R = T^k \vdash S^k$ such that $inf_R \neq r_T$. Lemma 4.21 then implies the label λ_k is $(k)_{crit\left(r_T, T^k \vdash S^k\right)}$. Third, we prove $crit\left(r_T, T^k \vdash S^k\right) = crit\left(\lambda_T^k\right)$.

We first show that tree S^k is a branch of canonical backbone B_{T^k} in tree $T^k \vdash S^k$, and then prove there is no critical vertex inf_R in tree $R = T^k \vdash S^k$ such that

$\mathit{inf}_R \neq r_T$. Since $\left|\lambda_T^k\right| = 1$ by the fact that k is the only element of λ_T^k, Lemma 4.32 implies B_{T^k} contains root r_T. But then S^k is a branch of B_{T^k} in $T^k \vdash S^k$, because $T^k \vdash S^k$ is formed from tree T^k by attaching tree S^k to r_T (Figure 5.5). By Claim

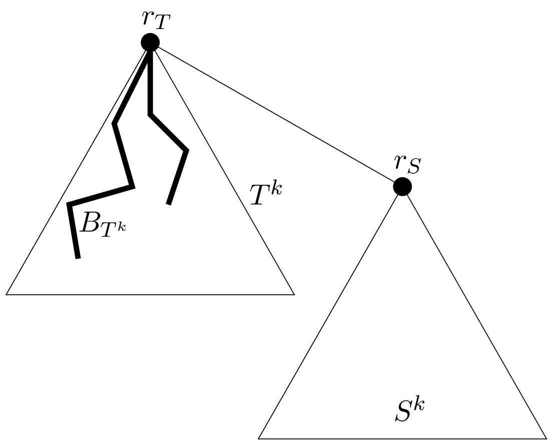

Figure 5.5: The proof of Lemma 5.8.

5.3, if inf_R is a critical vertex in $T^k \vdash S^k$ such that $\mathit{inf}_R \neq r_T$, then $\mathrm{VS}\left(T^k\right) = k$ and inf_R is a critical vertex in T^k, or $\mathrm{VS}\left(S^k\right) = k$ and inf_R is a critical vertex in S^k. The second possibility cannot be true, because $\mathrm{VS}\left(S^k\right) < k$. If the first possibility is true, then it follows from Lemma 4.23 and the fact $\left|\lambda_T^k\right| = 1$ that $\mathit{inf}_R = r_T$. We conclude that if $T^k \vdash S^k$ has a critical vertex inf_R, then $\mathit{inf}_R = r_T$.

Finally, we prove that the criticality of root r_T in tree $T^k \vdash S^k$ is $\mathit{crit}\left(\lambda_T^k\right)$. Since the subtrees yielded by r_T in $T^k \vdash S^k$ are tree S^k and the subtrees yielded by r_T in tree T^k, we conclude from the facts $k = \mathrm{VS}\left(T^k \vdash S^k\right) = \mathrm{VS}\left(T^k\right)$ and $\mathrm{VS}\left(S^k\right) < k$ that $\mathit{crit}\left(r_T, T^k \vdash S^k\right) = \mathit{crit}\left(r_T, T^k\right)$. Because k is the only element of label λ_T^k, Lemma 4.21 implies $\mathit{crit}\left(\lambda_T^k\right) = \mathit{crit}\left(r_T, T^k\right)$. We conclude that $\mathit{crit}\left(r_T, T^k \vdash S^k\right) = \mathit{crit}\left(\lambda_T^k\right)$. □

In the second lemma of the case analysis, we analyze the case when k is the only element of label λ_S^k and is noncritical (line 2 of algorithm `ADD-LABEL-UNEQUAL`).

Lemma 5.9 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that $\lambda_T^k(1) \neq \lambda_S^k(1)$ and $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$. If k is the only element of λ_S^k and is noncritical, then the label of tree $T^k \vdash S^k$ is $(k)_1$.

Proof. The structure of the proof is similar to that of Lemma 5.8. We prove the lemma by first constructing a k-backbone of tree $T^k \vdash S^k$ from the canonical backbone B_{S^k} of tree S^k. This implies by Lemmas 2.1 and 4.3 that since $\lambda_S^k(1) = \mathrm{VS}\left(S^k\right) = k$ (Lemma 4.19) and S^k is a subtree of $T^k \vdash S^k$, the vertex separation of tree $T^k \vdash S^k$ is k. Second, we show that there is no critical vertex inf_R in $R = T^k \vdash S^k$ such that $\mathit{inf}_R \neq r_T$, implying by Lemma 4.21 that label λ_k is $(k)_{crit\left(r_T, T^k \vdash S^k\right)}$. Third, we prove that the criticality of r_T in $T^k \vdash S^k$ is 1.

We first show that tree $T^k \vdash S^k$ has a k-backbone by considering the canonical backbone B_{S^k} of tree S^k. Since k is the only element of label λ_S^k and is noncritical, $\left|\lambda_S^k\right| = 1$ holds and λ_S^k is a noncritical label. Therefore, Lemma 4.32 implies the canonical backbone B_{S^k} of tree S^k has as one endpoint the root r_S of S^k. Since the concept of the canonical backbone is independent of its orientation, we may assume without loss of generality that r_T is the left endpoint of B_{S^k}. Because the root r_T of tree T^k is adjacent to r_S in $T^k \vdash S^k$, we may consider the path $P = r_T + B_{S^k}$ in $T^k \vdash S^k$. Every branch of P in $T^k \vdash S^k$ is either a branch of r_T in T^k or a branch of B_{S^k} in S^k; that is, $\left(T^k \vdash S^k\right)[P] = T^k[r_T] \cup S^k\left[B_{S^k}\right]$ (Figure 5.6). The facts

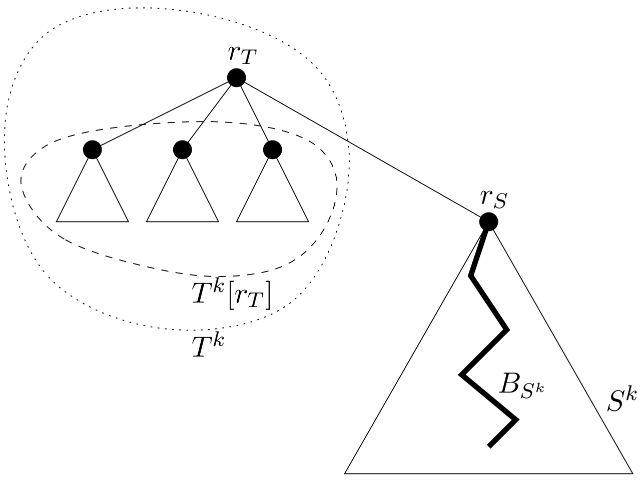

FIGURE 5.6: The proof of Lemma 5.9.

that $\mathrm{VS}\left(T^k\right) = \lambda_T^k(1) \neq \lambda_S^k(1) = \mathrm{VS}\left(S^k\right)$ and $k = \lambda_S^k(1) = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$ imply $\mathrm{VS}\left(T^k\right) < k$. But then Lemma 2.1 implies all subtrees in $T^k[r_T]$ have vertex separation less than k. Hence, since all subtrees in $S^k\left[B_{S^k}\right]$ have vertex separation less than k by the fact that B_{S^k} is a backbone of S^k, it follows that all subtrees in set $\left(T^k \vdash S^k\right)[P]$ have vertex separation less than k. Hence, path P is a k-backbone of tree $T^k \vdash S^k$.

We next prove that there is no critical vertex inf_R in tree $R = T^k \vdash S^k$ such that $\mathit{inf}_R \neq r_T$. If there is a critical vertex inf_R in $T^k \vdash S^k$, then Claim 5.3 yields $\mathit{inf}_R = r_T$, because VS $\left(T^k\right) < k$ and Lemma 4.24 implies there is no critical vertex in tree S^k.

Third, we show the criticality of label r_T in $T^k \vdash S^k$ is 1. Lemma 2.1 and the fact VS $\left(T^k\right) < k$ imply all subtrees yielded by r_T in tree T^k have vertex separation less than k. Furthermore, each subtree yielded by r_T in $T^k \vdash S^k$ is either a subtree yielded by r_T in T^k, or it is the tree S^k. Together with the fact that VS $\left(S^k\right) = k$, we conclude that r_T yields exactly one subtree, S^k, with vertex separation k in $T^k \vdash S^k$. We have thus shown that $\mathit{crit}\left(r_T, T^k \vdash S^k\right) = 1$. □

The third case of our case analysis applies when neither of the first two cases (Lemmas 5.8 and 5.9) applies and $\lambda_{k-1}(1) < k$, where λ_{k-1} is the label of tree $T^{k-1} \vdash S^{k-1}$ (line 3 of algorithm `ADD-LABEL-UNEQUAL`). This condition is equivalent to saying that k is a critical element of either label λ_T^k or label λ_S^k, there is an element of λ_T^k smaller than k, and $\lambda_{k-1}(1) < k$; this will be shown later in the correctness proof of the algorithm (Lemma 5.12).

Lemma 5.10 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that $\lambda_T^k(1) \neq \lambda_S^k(1)$ and $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$, and suppose λ_{k-1} is the label of tree $T^{k-1} \vdash S^{k-1}$. If k is a critical element of either λ_T^k or λ_S^k, there is an element of λ_T^k smaller than k, and $\lambda_{k-1}(1) < k$, then the label of tree $T^k \vdash S^k$ is label λ_{k-1} prepended with k.

Proof. We initially observe that since there is an element of λ_T^k smaller than k, point 1 in Lemma 4.35 implies tree T^{k-1} is not empty. Therefore, tree $T^{k-1} \vdash S^{k-1}$ is nonempty, and label λ_{k-1} is defined. We prove the lemma by first showing that the vertex separation of tree $T^k \vdash S^k$ is k and then applying Lemma 4.31.

We first show VS $\left(T^k \vdash S^k\right) = k$ by demonstrating that the canonical backbone of either tree T^k or tree S^k is a k-backbone of tree $T^k \vdash S^k$. We denote by inf_{X^k} the critical vertex in tree X^k, where $X^k = T^k$ if k is a critical element of λ_T^k and

$X^k = S^k$ if k is a critical element of λ_S^k; that is, inf_{X^k} is the vertex in tree X^k corresponding to the (first) element k of label λ_X^k.

We now consider the canonical backbone B_{X^k} of tree X^k, and show that all branches of B_{X^k} in tree $T^k \vdash S^k$ have vertex separation less than k. We analyze two cases, depending on whether $\left|\lambda_X^k\right| > 1$ or $\left|\lambda_X^k\right| = 1$.

We first consider the case when $\left|\lambda_X^k\right| > 1$. Since inf_{X^k} is the vertex in tree X^k corresponding to the first element of λ_X^k, Lemma 4.33 implies B_{X^k} is nonmonotonic, inf_{X^k} is the inflection vertex of B_{X^k}, and one of the branches of B_{X^k} in X^k is the subtree $X^k\langle inf_{X^k}\rangle$. Thus, every vertex u in B_{X^k} is a descendant in X^k of inf_{X^k}, and Lemma 5.1 implies $\left(T^k \vdash S^k\right)_u = \left(X^k\right)_u$. Hence, every branch Y of B_{X^k} in X^k such that $Y \neq X^k\langle inf_{X^k}\rangle$ is a branch of B_{X^k} in $T^k \vdash S^k$, and every branch Y of B_{X^k} in $T^k \vdash S^k$ such that $Y \neq \left(T^k \vdash S^k\right)\langle inf_{X^k}\rangle$ is a branch of B_{X^k} in X^k. In other words,

$$\left(T^k \vdash S^k\right)[B_{X^k}] = \left(X^k[B_{X^k}] - \left\{X^k\langle inf_{X^k}\rangle\right\}\right) \cup \left\{\left(T^k \vdash S^k\right)\langle inf_{X^k}\rangle\right\}.$$

Figures 5.7 and 5.8 illustrate the cases $X^k = T^k$ and $X^k = S^k$, respectively. Claim 5.4 implies

$$T^{k-1} \vdash S^{k-1} = \left(T^k \vdash S^k\right)\langle inf_{X^k}\rangle. \tag{5.2}$$

It follows from the facts that B_{X^k} is a backbone of tree X^k and $\text{VS}\left(X^k\right) = \lambda_X^k(1) = k$ (Lemma 4.19) that every subtree in set $X^k[B_{X^k}]$ has vertex separation less than k. And since $\text{VS}\left(T^{k-1} \vdash S^{k-1}\right) = \lambda_{k-1}(1) < k$, we conclude from Equation 5.2 that $\text{VS}\left(\left(T^k \vdash S^k\right)\langle inf_{X^k}\rangle\right) < k$, and hence every branch of B_{X^k} in tree $T^k \vdash S^k$, that is, every branch in set $\left(T^k \vdash S^k\right)[B_{X^k}]$, has vertex separation less than k. We next derive the identical conclusion for the second case, and then combine the two cases.

We now analyze the case when $\left|\lambda_X^k\right| = 1$. If $X^k = T^k$, then λ_T^k contains element k by the definition of tree X^k, and it also contains an element smaller than k, which is one of the suppositions. Hence, $\left|\lambda_T^k\right| > 1$, which is a contradiction of the assumption $\left|\lambda_X^k\right| = 1$. Therefore, $X^k = S^k$ holds. By Lemma 4.32, the canonical backbone B_{S^k} of S^k contains the root r_S of S^k. Since every vertex u in B_{S^k} is in tree S^k, Lemma 5.1 implies $\left(T^k \vdash S^k\right)_u = \left(S^k\right)_u$, which means that every branch

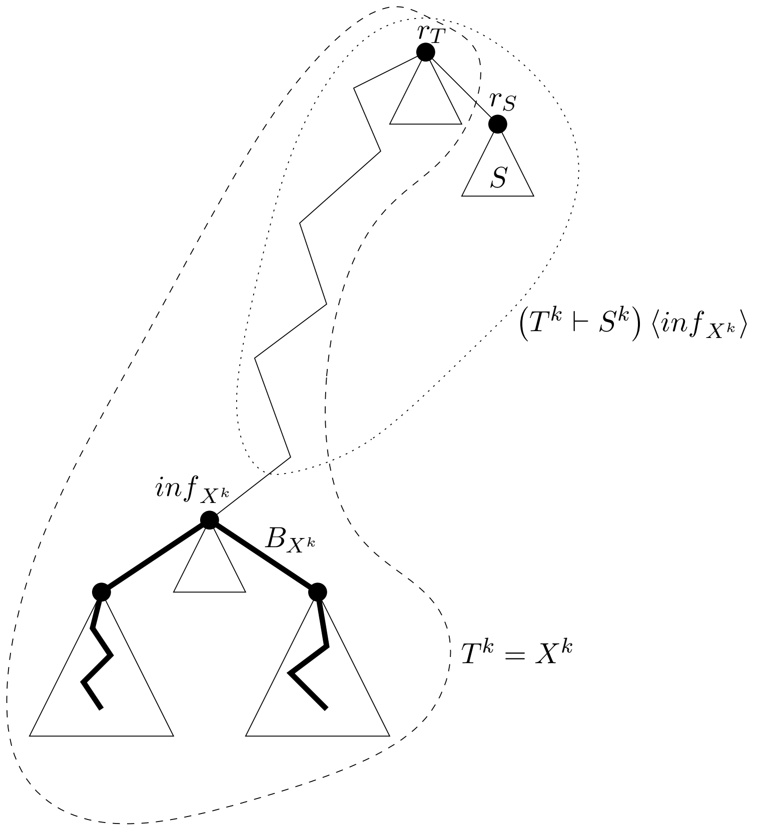

FIGURE 5.7: The proof of Lemma 5.10 when $\left|\lambda_X^k\right| > 1$ and $X^k = T^k$.

of B_{S^k} in S^k is a branch of B_{S^k} in $T^k \vdash S^k$. By the definition of tree $T^k \vdash S^k$ and the fact that r_S is the root of tree S^k, $\left(T^k \vdash S^k\right)\langle r_S\rangle = T^k$ holds. It follows that $\left(T^k \vdash S^k\right)[B_{S^k}] = S^k[B_{S^k}] \cup \left\{T^k\right\}$ (Figure 5.9), since r_S is in B_{S^k}. Because VS $\left(S^k\right) = k$, VS $\left(T^k\right) < k$. Therefore, since the fact that B_{S^k} is a backbone of tree S^k implies every subtree in set $S^k[B_{S^k}]$ has vertex separation less than k, every subtree in set $\left(T^k \vdash S^k\right)[B_{S^k}]$ has vertex separation less than k. We remark that Equation 5.2 holds in this case as well.

Combining cases $\left|\lambda_X^k\right| > 1$ and $\left|\lambda_X^k\right| = 1$, we conclude that all branches in tree $T^k \vdash S^k$ of the canonical backbone B_{X^k} of tree X^k have vertex separation less than k, and thus Lemma 4.3 yields VS $\left(T^k \vdash S^k\right) \leq k$. Lemma 2.1 and the facts that X^k is a subtree of $T^k \vdash S^k$ and $k = \lambda_X^k(1) =$ VS $\left(X^k\right)$ imply VS $\left(T^k \vdash S^k\right) \geq k$. Hence, VS $\left(T^k \vdash S^k\right) = k$ holds.

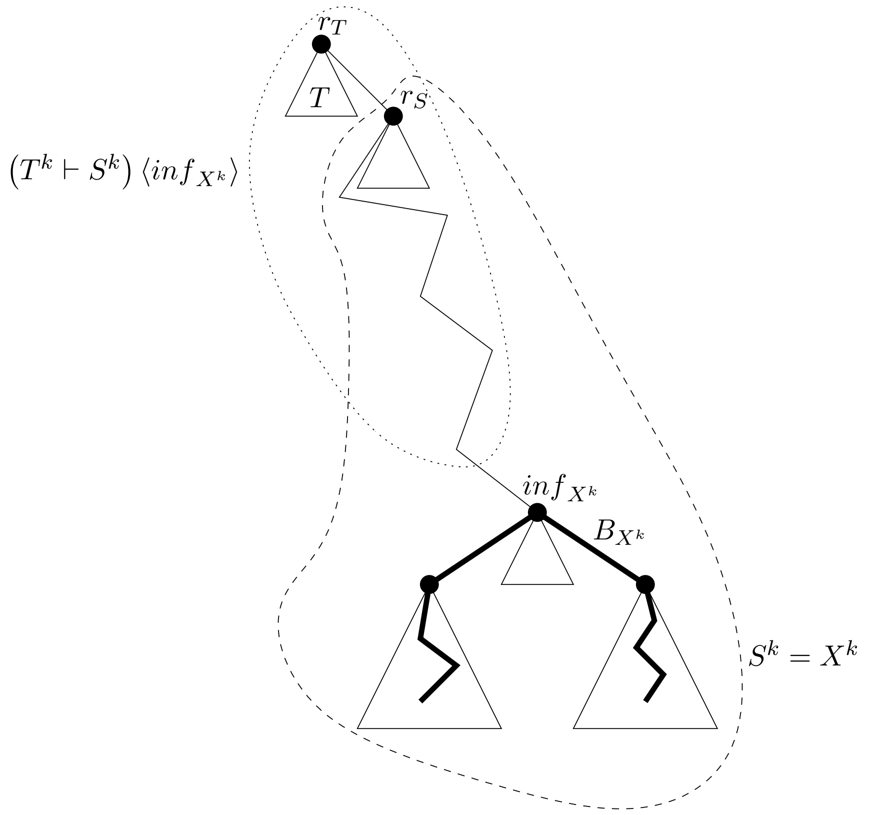

FIGURE 5.8: The proof of Lemma 5.10 when $\left|\lambda_X^k\right| > 1$ and $X^k = S^k$.

We conclude the proof by constructing the label λ_k of tree $T^k \vdash S^k$ using Lemma 4.31. We first show that $inf_{X^k} \neq r_T$. If $X^k = S^k$, then $inf_{X^k} \neq r_T$, since r_T is not in tree S^k. If $X^k = T^k$, then label λ_T^k contains at least two elements: k and an element smaller than k. Lemma 4.27 therefore implies the vertex inf_{X^k}, which corresponds to the first element k of λ_T^k, is not the root r_T. Thus, in either case, $inf_{X^k} \neq r_T$. The critical vertex inf_{X^k} of X^k is also critical in subtree $\left(X^k\right)_b$, where $b = inf_{X^k}$. Since $inf_{X^k} \neq r_T$, Lemma 5.1 implies $\left(X^k\right)_b = \left(T^k \vdash S^k\right)_b$. Thus, the root inf_{X^k} of tree $\left(T^k \vdash S^k\right)_b$ is critical in the tree, and Lemma 2.1 implies VS $\left(\left(T^k \vdash S^k\right)_b\right) = k$. It follows from Equation 5.2 that VS $\left(\left(T^k \vdash S^k\right) \langle inf_{X^k} \rangle\right) =$ VS $\left(T^{k-1} \vdash S^{k-1}\right) = \lambda_{k-1}(1) < k$. We conclude from Lemma 4.31 that the label of tree $T^k \vdash S^k$ is label λ_{k-1} prepended with k, since $T^k \vdash S^k = \left(T^k \vdash S^k\right) \langle inf_{X^k} \rangle \vdash_p \left(T^k \vdash S^k\right)_b$, where p is the parent of vertex $inf_{X^k} = b$. $\square$

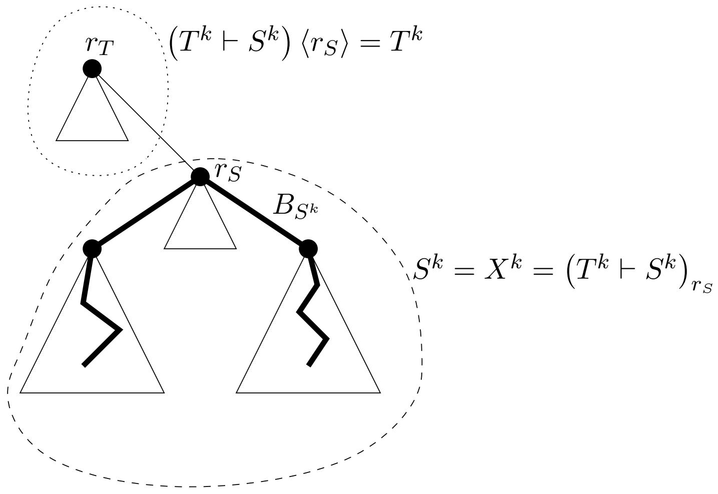
r_T
$\left(T^k \vdash S^k\right)\langle r_S\rangle = T^k$
r_S
B_{S^k}
$S^k = X^k = \left(T^k \vdash S^k\right)_{r_S}$

FIGURE 5.9: The proof of Lemma 5.10 when $\left|\lambda_X^k\right| = 1$.

Finally, we consider the case when none of the first three cases applies (line 4 of algorithm `ADD-LABEL-UNEQUAL`). The only difference between this case and the third case (Lemma 5.10) is that $\lambda_{k-1}(1) \geq k$ in this case, as opposed to $\lambda_{k-1}(1) < k$ in Lemma 5.10.

Lemma 5.11 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that $\lambda_T^k(1) \neq \lambda_S^k(1)$ and $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$, and suppose λ_{k-1} is the label of tree $T^{k-1} \vdash S^{k-1}$. If k is a critical element of either λ_T^k or λ_S^k, there is an element of λ_T^k smaller than k, and $\lambda_{k-1}(1) \geq k$, then the label of tree $T^k \vdash S^k$ is $(k+1)_0$.

Proof. Like at the beginning of the proof of Lemma 5.10, we initially note that since there is an element of label λ_T^k smaller than k, point 1 in Lemma 4.35 guarantees tree T^{k-1} is nonempty. Therefore, tree $T^{k-1} \vdash S^{k-1}$ is nonempty and label λ_{k-1} is defined. We prove the lemma by showing that $\mathrm{VS}\left(T^k \vdash S^k\right) > k$. Since $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\} = \max\left\{\mathrm{VS}\left(T^k\right), \mathrm{VS}\left(S^k\right)\right\}$ (Lemma 4.19), Claim 5.2 then implies the label λ_k of tree $T^k \vdash S^k$ is $(k+1)_0$.

In order to prove $\mathrm{VS}\left(T^k \vdash S^k\right) > k$, we first show that $\mathrm{VS}\left(\left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle\right) \geq k$, where X^k is the tree T^k or S^k depending on whether k is a critical element of label λ_T^k or λ_S^k. By Claim 5.4, $T^{k-1} \vdash S^{k-1} = \left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle$ holds. Since $\mathrm{VS}\left(T^{k-1} \vdash S^{k-1}\right) = \lambda_{k-1}(1) \geq k$, we conclude that $\mathrm{VS}\left(\left(T^k \vdash S^k\right)\langle \mathit{inf}_{X^k}\rangle\right) \geq k$.

We next show that inf_{X^k} yields two subtrees in tree $T^k \vdash S^k$ with vertex separation k, and conclude that $\mathrm{VS}\left(T^k \vdash S^k\right) > k$. Toward this goal, we first prove $inf_{X^k} \neq r_T$. We consider two cases, depending on whether $X^k = S^k$ or $X^k = T^k$. If $X^k = S^k$, then clearly $inf_{X^k} \neq r_T$. If $X^k = T^k$, the label λ_T^k contains at least two elements: k and an element smaller than k. But then $\left|\lambda_T^k\right| > 1$, and Lemma 4.27 implies $inf_{X^k} \neq r_T$, because inf_{X^k} corresponds to the first element of label λ_T^k. Therefore, in both cases, $inf_{X^k} \neq r_T$. Vertex inf_{X^k} yields two subtrees X_1 and X_2 in tree X^k with vertex separation k, and hence it follows from Lemma 5.1 that it yields the same two subtrees in tree $T^k \vdash S^k$. This implies vertex inf_{X^k} has three branches in $T^k \vdash S^k$ with vertex separation at least k: $\left(T^k \vdash S^k\right)\langle inf_{X^k}\rangle$, X_1, and X_2. We conclude from Lemma 4.4 that $\mathrm{VS}\left(T^k \vdash S^k\right) > k$. □

The correctness of algorithm `ADD-LABEL-UNEQUAL` (Algorithm 5.2) is a consequence of Lemmas 5.8, 5.9, 5.10, and 5.11.

Lemma 5.12 Suppose we have trees T^k and S^k with labels λ_T^k and λ_S^k, respectively, such that $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$, and suppose λ_{k-1} is the label of tree $T^{k-1} \vdash S^{k-1}$; if $T^{k-1} \vdash S^{k-1}$ is empty, then λ_{k-1} is undefined and is not used by Algorithm 5.2. If $\lambda_T^k(1) \neq \lambda_S^k(1)$, then on input $\left(\lambda_T^k, \lambda_S^k, \lambda_{k-1}\right)$ Algorithm 5.2 correctly computes the label λ_k of tree $T^k \vdash S^k$.

Proof. We prove the lemma by showing that the algorithm executes line 1, 2, 3, or 4 if and only if the condition of Lemma 5.8, 5.9, 5.10, or 5.11, respectively, is satisfied. Lemma 5.12 then follows by observing that the algorithm sets the return label λ_k to the correct label in each of the four cases. We also show that if label λ_{k-1} is undefined, then one of the first two lines of the algorithm is executed, and therefore argument λ_{k-1} is not used.

We first show that the conditions on lines 1 and 2 of the algorithm correspond directly to the conditions of Lemmas 5.8 and 5.9, respectively. The conditions $\lambda_T^k(1) \neq \lambda_S^k(1)$, $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$, and λ_{k-1} being the label of tree $T^{k-1} \vdash S^{k-1}$ are common to Lemmas 5.8, 5.9, 5.10, and 5.11, and they are also preconditions of the algorithm. Hence, we focus on the other, more specific conditions. If k is the only element of λ_T^k, then the algorithm sets λ_k to $(k)_{crit\left(\lambda_T^k\right)}$ on line 1, which is correct (Lemma 5.8). If k is the only element of λ_S^k and is noncritical, then k is not an element of label λ_T^k by the facts that $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$ and $\lambda_T^k(1) \neq \lambda_S^k(1)$. Hence, the condition on line 1 of the algorithm cannot be true, and the algorithm sets λ_k to $(k)_1$ on line 2, which is correct (Lemma 5.9).

Next, we show that if neither of the first two cases on lines 1 and 2 applies, then k is a critical element of either λ_T^k or λ_S^k and there is an element of label λ_T^k smaller than k. This is equivalent to the common conditions of Lemmas 5.10 and 5.11. If neither of the first two cases applies, then k is neither the only element of λ_T^k nor the only element of λ_S^k that is noncritical. There are thus three possible cases. In the first case, k is an element of λ_T^k and $\left|\lambda_T^k\right| > 1$. In the second case, k is an element of λ_S^k and $\left|\lambda_S^k\right| > 1$. In the third case, k is the only element of λ_S^k and is critical. We show that in each case, k is a critical element of either λ_T^k or λ_S^k and there is an element of label λ_T^k smaller than k.

First, we consider the case when k is an element of λ_T^k and $\left|\lambda_T^k\right| > 1$. Lemma 4.30 implies k is a critical element of label λ_T^k, since k is the first element of λ_T^k. We also infer from Lemma 4.20 and the fact $\left|\lambda_T^k\right| > 1$ that there is an element of λ_T^k smaller than k.

Second, we consider the case when k is an element of λ_S^k and $\left|\lambda_S^k\right| > 1$. Like in the previous case, it follows from Lemma 4.30 that k is a critical element of λ_S^k. Since $\lambda_T^k(1) \neq \lambda_S^k(1)$, $\lambda_T^k(1) < k$ holds, and therefore the element $\lambda_T^k(1)$ of label λ_T^k is smaller than k.

Third, we consider the case when k is the only element of λ_S^k and is critical. We only need to observe that there is an element of λ_T^k smaller than k, since $\lambda_T^k(1) < k$.

We conclude the proof by showing that lines 3 and 4 of the algorithm correctly compute the label λ_k of tree $T^k \vdash S^k$, assuming neither of the cases on lines 1 and 2 applies. We remark that since there is an element of λ_T^k smaller than k, point 1 in Lemma 4.35 implies tree T^{k-1} is not empty. Thus, tree $T^{k-1} \vdash S^{k-1}$ is nonempty and label λ_{k-1} is defined. Therefore, if λ_{k-1} is undefined, one of the first two cases on lines 1 and 2 applies and the algorithm does not use λ_{k-1}. If $\lambda_{k-1}(1) < k$, then the algorithm executes line 3 and sets λ_k to label λ_{k-1} prepended with k, which is correct (Lemma 5.10). If $\lambda_{k-1}(1) \geq k$, then line 4 is executed and λ_k is set to $(k+1)_0$; Lemma 5.11 implies this is correct. □

Having shown the correctness of algorithms `ADD-LABEL-EQUAL` and `ADD-LABEL-UNEQUAL`, we now prove the correctness and running time of the main algorithm, algorithm `ADD-LABEL` (Algorithm 5.3).

Theorem 5.13 Suppose we have trees T and S with labels λ_T and λ_S. Then on input (λ_T, λ_S) Algorithm 5.3 correctly computes the label λ of tree $T \vdash S$.

Proof. We prove the theorem by induction on the number i of iterations of the loop on lines 5–9. The technique we employ is similar to that used in the correctness proof of algorithm `COMBINE-LABELS` (Algorithm 4.1) [EST94], which was described starting on page 59. The loop in algorithm `ADD-LABEL` iterates from $m = \max\{1, \lambda_T(|\lambda_T|)\}$ to $M = \max\{\lambda_T(1), \lambda_S(1)\}$, however, and therefore during the ith iteration, $1 \leq i \leq M - m + 1$, the relationship between i and the loop variable k is

$$k = i + m - 1. \tag{5.3}$$

We show that λ is the correct label of the tree $T^k \vdash S^k$ just after the ith iteration is completed.

Before we start the induction, there is one technicality worth mentioning at this point. If $\lambda_T(|\lambda_T|) > 0$, then $m = \max\{1, \lambda_T(|\lambda_T|)\} = \lambda_T(|\lambda_T|)$. By point 1 in Lemma 4.35, this implies tree T^{m-1} is empty, and therefore tree $T^{m-1} \vdash S^{m-1}$ is empty. Thus, label λ is undefined when $i = 0$ (base case). However, because m is at least the smallest element of λ_T, at the end of the ith iteration of the loop, where $1 \leq i \leq M - m + 1$, point 1 in Lemma 4.35 implies tree $T^k = T^{i+m-1}$ is not empty, and hence tree $T^k \vdash S^k = T^{i+m-1} \vdash S^{i+m-1}$ is nonempty. Nevertheless, since the induction step covers the case when $i = 1$, it is possible that the label λ used in the induction hypothesis is undefined, since it is the label of tree $T^{m-1} \vdash S^{m-1}$, which can be empty. We will see that in this case, the label of $T^{m-1} \vdash S^{m-1}$ is not used in the induction step.

We now prove the base case of the induction. There are two cases, depending on whether $\lambda_T(|\lambda_T|) > 0$ or $\lambda_T(|\lambda_T|) = 0$. If $\lambda_T(|\lambda_T|) > 0$, then tree $T^{m-1} \vdash S^{m-1}$ is empty; this was just shown in the previous paragraph. Thus, label λ is undefined. The algorithm does not set λ to any value before entering the loop, because the condition on line 1 is false.

We now turn to the case when $\lambda_T(|\lambda_T|) = 0$. In this case, $m = \max\{1, 0\} = 1$, and thus our objective is to show that λ is the label of tree $T^{m-1} \vdash S^{m-1} = T^0 \vdash S^0$ before the algorithm enters the loop. In this case, the condition on line 1 is true, and the algorithm executes one of the two assignment statements on lines 2 and 3. By point 1 in Lemma 4.35, tree T^0 is not empty and the largest element of the label λ_T^0 of T^0 is 0 (point 2 in Lemma 4.35). Lemma 4.20 therefore implies $\lambda_T^0 = (0)_{crit(\lambda_T^0)}$. Hence, Lemma 4.19 yields $\mathrm{VS}(T^0) = 0$, and Lemma 2.2 implies T^0 is the trivial tree.

We now consider two subcases, depending on whether $\lambda_S(|\lambda_S|) > 0$ or $\lambda_S(|\lambda_S|) = 0$. If $\lambda_S(|\lambda_S|) > 0$, then we conclude from point 1 in Lemma 4.35 that tree S^0 is empty. Therefore, $T^0 \vdash S^0 = T^0$, and hence $\lambda = \lambda_T^0 = (0)_0$; the criticality of the only vertex in T^0 is clearly 0. In this case, the algorithm correctly sets λ to $(0)_0$ on line 2. If $\lambda_S(|\lambda_S|) = 0$, then point 1 in Lemma 4.35 implies tree S^0 is not empty and its label is $\lambda_S^0 = (0)_{crit(\lambda_S^0)}$. It follows from Lemma 2.2 that S^0 is the trivial tree, and thus $T^0 \vdash S^0$ is the path graph P_2. By Lemma 2.3, the vertex separation of P_2 is 1. There are clearly no critical vertices in $T^0 \vdash S^0$, and the root r_T of $T^0 \vdash S^0$ yields only one subtree, the trivial tree S^0, whose vertex separation is 0. Therefore, the criticality of r_T in $T^0 \vdash S^0$ is 0, and hence the criticality of the label of $T^0 \vdash S^0$ is 0. We conclude that $\lambda = (1)_0$. Label λ is correctly set to this value on line 3 of the algorithm. This completes our analysis of the base case.

We prove the induction step by showing that at the end of the ith iteration, $1 \leq i \leq M - m + 1$, λ is the correct label of tree $T^k \vdash S^k$, where $k = i + m - 1$ (Equation 5.3). We assume λ is the correct label of tree $T^{k-1} \vdash S^{k-1}$ just before the ith iteration begins; this is our induction hypothesis. We will prove the induction step shortly. Since the loop terminates right after the $(M - m + 1)$th iteration, λ is the label of tree $T^M \vdash S^M$ after the algorithm finishes execution. It follows from the facts $M \geq \lambda_T(1)$ and $M \geq \lambda_S(1)$ that $T^M = T$ and $S^M = S$ (the first case in Definition 4.5). Thus, λ is the label of tree $T \vdash S$ after the algorithm exits.

The induction step uses the correctness of the two subroutines called by the main algorithm, `ADD-LABEL-EQUAL` and `ADD-LABEL-UNEQUAL`, proved in Lemmas 5.7 and 5.12, respectively. If k is an element of both λ_T and λ_S, then by point 2 in Lemma 4.35 and the fact that a label is a strictly decreasing sequence (Lemma

4.20), the first element of both λ_T^k and λ_S^k is k; that is, $k = \lambda_T^k(1) = \lambda_S^k(1)$. Hence, all the conditions of Lemma 5.7 are satisfied, and subroutine `ADD-LABEL-EQUAL` (Algorithm 5.1), called on line 7, correctly computes the label λ of tree $T^k \vdash S^k$. If k is an element of exactly one of λ_T and λ_S, then it follows from Lemmas 4.20 and 4.35 that the first element of exactly one of the labels λ_T^k and λ_S^k is k; that is, $\lambda_T^k(1) \neq \lambda_S^k(1)$ and $k = \max\left\{\lambda_T^k(1), \lambda_S^k(1)\right\}$. Therefore, all the conditions of Lemma 5.12 are satisfied, and subroutine `ADD-LABEL-UNEQUAL` (Algorithm 5.2), called on line 9, correctly computes the label λ of tree $T^k \vdash S^k$, assuming λ is the correct label of tree $T^{k-1} \vdash S^{k-1}$, which is true by the induction hypothesis. We note that $T^{k-1} \vdash S^{k-1}$ can be empty, in which case λ is undefined.

Finally, if k is not an element of either λ_T or λ_S, then $T^k = T^{k-1}$ and $S^k = S^{k-1}$ by the following argument. If $k > \lambda_X(1)$, where $X = T$ or $X = S$, then $k - 1 \geq \lambda_X(1)$, and by the base case of the recursive definition of X^{k-1}, $X^{k-1} = X = X^k$ holds. If $k \leq \lambda_X(1)$, then $X^{k-1} = X^k$, since k is not an element of λ_X (the second case in Definition 4.5). Therefore, $T^k \vdash S^k = T^{k-1} \vdash S^{k-1}$, which implies the labels of trees $T^k \vdash S^k$ and $T^{k-1} \vdash S^{k-1}$ are the same. The iteration correctly does not modify λ in this case. □

Theorem 5.14 Algorithm 5.3 can be implemented to run in $O(\mathrm{VS}(T \vdash S)) = O(\lg |T \vdash S|)$ time on input (λ_T, λ_S), where λ_T and λ_S are the labels of trees T and S.

Proof. We prove the theorem by showing how to achieve constant running time per iteration of the loop on lines 5–9 of the algorithm. This, together with the fact that the loop iterates at most M times, implies the running time of the algorithm is $O(M)$; lines 1–4 are simple operations that can be implemented in $O(1)$ time. Since $M \leq \mathrm{VS}(T)$ and $M \leq \mathrm{VS}(S)$, and because both trees T and S are subtrees of $T \vdash S$, we conclude from Lemma 2.1 that $M \leq \mathrm{VS}(T \vdash S)$. Lemma 4.6 then implies the running time of the algorithm is $O(\mathrm{VS}(T \vdash S)) = O(\lg |T \vdash S|)$.

We first describe how to achieve constant running time for subroutines `ADD-LA-BEL-EQUAL` (Algorithm 5.1) and `ADD-LABEL-EQUAL` (Algorithm 5.2). Algorithm 5.1 consists of simple operations and can be readily implemented to run in $O(1)$ time. Algorithm 5.2 also consists of simple constant-time operations, except for the prepending operation on line 3. If the condition on line 3 is true, then λ_k is assigned label λ_{k-1} prepended with k. Since the argument λ_{k-1} is not used again in Algorithm 5.2, we can simply prepend λ_{k-1} with k (a constant-time operation), and then assign to λ_k the pointer to λ_{k-1}. In this way, we avoid copying λ_{k-1} over to λ_k, which is not, in general, a constant-time operation. We also pass argument λ_{k-1} and return label λ_k by means of a pointer; this is justified, because variable λ used in algorithm `ADD-LABEL` and passed to subroutine `ADD-LABEL-UNEQUAL` as λ_{k-1} on line 9 is immediately assigned to the value returned by the subroutine, and hence its former value is not needed again in algorithm `ADD-LABEL`.

We next show how to implement the calls on lines 7 and 9 of algorithm `ADD-LABEL` in $O(1)$ time. Since it follows from point 2 in Lemma 4.35 that labels λ_T^k and λ_S^k are contiguous subsequences of labels λ_T and λ_S, respectively, we can specify λ_T^k and λ_S^k by two pairs of integers delimiting the subsequences of λ_T and λ_S, together with unmodified versions of λ_T and λ_S. We pass labels λ_T and λ_S by means of a pointer to eliminate the need for copying them during the call. Since neither algorithm `ADD-LABEL-EQUAL` nor algorithm `ADD-LABEL-UNEQUAL` modifies the arguments λ_T^k and λ_S^k, passing the labels by means of a pointer is justified. We also pass by means of a pointer the argument λ on line 9, as discussed in the previous paragraph. □

So far in this chapter, we have discussed computing the label of a tree T as another tree S is attached to the root r_T of T. It is natural to ask the converse question: given a tree T with root r_T and label λ_T, and a child u of r_T with label $\lambda_{u,T} = \lambda_{T_u}$, what is the label $\lambda_{r_T,T\langle u\rangle} = \lambda_{T\langle u\rangle}$ after the subtree T_u is removed from T? Unfortunately, this question is impossible to answer given only labels λ_T and λ_{T_u}; to see this, consider the following example. Suppose that $\lambda_{T\langle u\rangle} = (k_1, k_2)_c$ and $\lambda_{T_u} = (k_1)_2$, where c is an integer satisfying $0 \leq c \leq 2$, and k_1 and k_2 are nonnegative integers satisfying $k_1 > k_2$. Then by Lemma 5.5 the label of $T = T\langle u\rangle \vdash T_u$ is $(k_1+1)_0$, regardless of the value of k_2. Hence, given only the labels $\lambda_T = (k_1+1)_0$ and $\lambda_{T_u} = (k_1)_2$ of trees T and T_u, it is not possible to compute the label of tree $T\langle u\rangle$. Chapter 6 describes how to augment the trees with extra information that makes it possible to recompute the label of tree T after a subtree yielded by root r_T is removed from T.

5.3 A Different Algorithm for Computing Vertex Separation

In this section, we show how the algorithm just given for computing the label of tree $T \vdash S$ from the labels of trees T and S can be used to find the vertex separation of a tree T. Our algorithm, called LABEL, runs in $O(|T| \lg |T|)$ time, which is the same time bound as that achieved by the Ellis-Sudborough-Turner algorithm (Theorem 4.46).

In algorithm LABEL, we view tree T as being composed of two nonempty subtrees T_1 and T_2 such that $T = T_1 \vdash T_2$. We first compute the labels λ_{T_1} of T_1 and λ_{T_2} of T_2 recursively, and then use algorithm ADD-LABEL on input $(\lambda_{T_1}, \lambda_{T_2})$ to compute the label, and hence vertex separation (Lemma 4.19), of tree T. Since the root of $T_1 \vdash T_2$ is the root of T_1, we have to choose trees T_1 and T_2 such that $T_1 = T\langle u\rangle$ and $T_2 = T_u$, where u is a child of the root r_T of T. Our algorithm chooses an arbitrary child u of r_T; it would be interesting to investigate if a more clever choice of u improves the running time of the algorithm (see Chapter 8 for an elaboration on this issue). Algorithm LABEL computes the vertex labelling of T as well.

Algorithm 5.4
Input: a tree T with root r_T.
Output: the vertex labelling of T.

```
LABEL($T, r_T$)
1.      if $r_T$ is a leaf then $\lambda_{r_T} \leftarrow (0)_0$
2.      else
3.          $u \leftarrow$ an arbitrary child of $r_T$
4.          LABEL($T\langle u\rangle, r_T$)
5.          LABEL($T_u, u$)
6.          $\lambda_{r_T} \leftarrow$ ADD-LABEL ($\lambda_{r_T}, \lambda_u$)
```

Both the correctness and running time of algorithm LABEL are simple results, and are stated and proved below.

Theorem 5.15 Suppose we have a tree T with root r_T. Then on input (T, r_T) Algorithm 5.4 correctly computes the label λ_v of each vertex v in T.

Proof. We prove the theorem by induction on the size n of tree T. The base case occurs when $n = 1$; that is, T is the trivial tree. By Lemma 2.2, $\mathrm{VS}(T) = 0$ holds. Furthermore, the root r_T of T is a leaf, and therefore its criticality is 0, because it does not yield any subtrees. Hence, the label of r_T is $(0)_0$. The algorithm correctly assigns this label to λ_{r_T} on line 1.

We now prove the induction step. We assume $n > 1$ and the algorithm works correctly for any tree of size at most $n - 1$. Since $|T\langle u\rangle| < n$ and $|T_u| < n$, where u is an arbitrary child of root r_T as assigned on line 3, the calls on lines 4 and 5 correctly compute the vertex labellings of trees $T\langle u\rangle$ and T_u, respectively. By Lemma 5.1, the only label of a vertex v in tree $T = T\langle u\rangle \vdash T_u$ that can be different from the label of v in tree X, where $X = T\langle u\rangle$ if v is in $T\langle u\rangle$ and $X = T_u$ if v is in T_u, is the label of the root r_T. But Theorem 5.13 implies the label λ_{r_T} is correctly updated by algorithm `ADD-LABEL` on line 6. $\square$

Theorem 5.16 Suppose we have a tree T with root r_T. Then on input (T, r_T) Algorithm 5.4 runs in time $O(|T|\mathrm{VS}(T)) = O(|T| \lg |T|)$.

Proof. We show the running time of the algorithm by first proving the bound of $O(|T|)$ on the total number of recursive calls made by the algorithm. Then, we show that the running time per a recursive call is $O(\mathrm{VS}(T)) = O(\lg |T|)$. These two claims imply the running time of the algorithm is $O(|T|\mathrm{VS}(T)) = O(|T| \lg |T|)$.

We first show that the total number of recursive calls made by the algorithm is $O(|T|)$. The pair of recursive calls on lines 4 and 5 can be thought of as corresponding to a removal of the edge $r_T u$ from tree T. Since the algorithm calls itself on each tree $T\langle u\rangle$ and T_u separately, no edge is removed more than once. Since there are $|T|-1$ edges in a tree on $|T|$ vertices, we conclude that the total number of recursive calls generated, including the initial call, is at most $2(|T| - 1) + 1 = O(|T|)$.

We now show that the running time per a recursive call is $O(\mathrm{VS}(T)) = O(\lg |T|)$. Lines 1 and 3 of the algorithm take $O(1)$ time. Lines 4 and 5 can be implemented in $O(1)$ time by simply updating a constant number of pointers to effectively remove edge $r_T u$ from tree T. Finally, it follows from Theorem 5.14 that the running time of line 6 is $O(\mathrm{VS}(T)) = O(\lg |T|)$, since $T = T\langle u\rangle \vdash T_u$. We conclude that the running time of one recursive call of the algorithm takes $O(\mathrm{VS}(T)) = O(\lg |T|)$ time. $\square$

Chapter 6

Removing a Subtree Yielded by the Root

In this chapter, we investigate how to quickly update the vertex separation of a tree T after a subtree T_{rm} yielded by the root r_T of T is removed (that is, vertex rm is a child of r_T). We present an $O\left(\text{VS}(T)^2\right)$ algorithm that updates the vertex labellings of trees $T\langle rm\rangle$ and T_{rm}. Since $T = T\langle rm\rangle \vdash T_{rm}$, Lemma 5.1 implies the only label of a vertex u in $T\langle rm\rangle$ that can be different from the label of u in tree T is label λ_{r_T}, and the label of every vertex in T_{rm} is the same as the label of that vertex in T. That is, our goal is to compute $\lambda_{r_T,T\langle rm\rangle}$. We denote this label by λ in order to avoid clutter. As discussed toward the end of Chapter 5 (page 92), it is not possible to find λ only from labels $\lambda_{r_T,T}$ and $\lambda_{rm,T}$, so an approach is needed that is different from that used in algorithm `ADD-LABEL` presented in that chapter.

In order to achieve the running time claimed, extra information is stored at vertices r_T and rm. This extra information in effect imposes a partial order on the children of r_T. We will discuss this extra information in Section 6.1. Although the need to keep the extra information has been motivated only for the root r_T of T, we will discuss it for an arbitrary vertex in T. This generalization will become important in Chapter 7, when we describe how to update the vertex labelling of a tree T after another tree is attached to an arbitrary vertex in T, or after a subtree yielded by an arbitrary vertex in T is removed from T.

The update algorithms presented in this chapter, which are `REMOVE-SUBTREE` and ATTACH-TREE, use a modified version of algorithm COMBINE-LABELS (Algorithm 4.1) as a subroutine, and by taking advantage of the extra information we improve its running time from $\Theta(dM)$ (Lemma 4.45) to $O(M^2)$, where d is the number of labels to combine and M is the maximum element of an input label. The algorithm, called FAST-COMBINE-LABELS, computes the label λ by combining the labels of all d children of r_T except rm. It is described in Section 6.2, together with algorithms REMOVE-SUBTREE and ATTACH-TREE.

6.1 VS-Ordered Trees and Strong Labelling

Toward the goal of improving the running time of algorithm COMBINE-LABELS from $\Theta(dM)$ to $O(M^2)$, we need to be able to store and efficiently update an ordering on the children of each vertex in tree T that is induced by the vertex separations of the subtrees rooted at the children. We call a tree T a VS-*ordered tree* if the children $u_1, \ldots, u_d$ of each vertex u in T are ordered so that $\mathrm{VS}\left(T_{u_1}\right), \ldots, \mathrm{VS}\left(T_{u_d}\right)$ is a monotonically decreasing sequence; that is, $1 \leq i < j \leq d$ implies $\mathrm{VS}(T_{u_i}) \geq \mathrm{VS}\left(T_{u_j}\right)$. A VS-ordered tree T is implemented by storing the pointers to the children $u_1, \ldots, u_d$ of each vertex u in a doubly linked list that is sorted in the order of nonincreasing value of $\mathrm{VS}(T_{u_i})$. From now on, we omit the mention of pointers when their use is implicit. For example, we say 'storing the children' instead of 'storing the pointers to the children.' Since we will need to argue about these doubly linked lists for different vertices and trees, we introduce notation to simplify the discussion.

Definition 6.1 Suppose that T is a rooted tree and u is a vertex in T whose children are $u_1, \ldots, u_d$. We denote by $\mathcal{D}_u^T$ the doubly linked list storing vertices $u_1, \ldots, u_d$ such that node α occurs before node β in the list if and only if α stores u_i, β stores u_j, and $\mathrm{VS}(T_{u_i}) \geq \mathrm{VS}\left(T_{u_j}\right)$. If vertex u is a leaf, then list $\mathcal{D}_u^T$ is empty.

We next explain how to update list $\mathcal{D}_u^T$ quickly after a subtree yielded by u is removed from tree T, or after a tree is attached to u. Without additional information, this can take linear time in the worst case, since the length of $\mathcal{D}_u^T$ can be $\Omega(|T|)$, and potentially we need to scan the entire list $\mathcal{D}_u^T$ to find the position of the root of the removed subtree (if a subtree has been removed), or to find a correct position for the root of the attached tree (if a tree has been attached to u). A faster, $O(\mathrm{VS}(T))$ running time is achieved by storing a few additional pieces of information at each vertex: arrays $\mathcal{A}$ and $\mathcal{Z}$, and pointer π. The pair of vertices $\mathcal{A}[k]$ and $\mathcal{Z}[k]$ delimits the start and end of the block of vertices in list $\mathcal{D}_u^T$ that are the roots of the subtrees yielded by u and having vertex separation k; this sublist of $\mathcal{D}_u^T$ is called the *k-block* of $\mathcal{D}_u^T$. With arrays $\mathcal{A}$ and $\mathcal{Z}$, it is possible to find the k-block in constant time. The pointer π is used to locate in constant time the node in $\mathcal{D}_{p_u}^T$ that stores the vertex u, where p_u is the parent of u. Figure 6.1 illustrates the arrays $\mathcal{A}$ and $\mathcal{Z}$, and the pointer π_u. In the figure, the 8-block of $\mathcal{D}_u^T$ consists of nodes 2–5.

Definition 6.2 Given a rooted tree T with the list $\mathcal{D}_u^T$ computed for a vertex u in T, we denote by $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$ arrays indexed from 0 to the largest vertex separation of a subtree of T yielded by u:

1. entry $\mathcal{A}_u^T[k]$ stores the first node in the k-block of list $\mathcal{D}_u^T$, and

2. entry $\mathcal{Z}_u^T[k]$ stores the last node in the k-block of $\mathcal{D}_u^T$.

If the k-block of $\mathcal{D}_u^T$ is empty, then $\mathcal{A}_u^T[k]$ and $\mathcal{Z}_u^T[k]$ are both null. If u is not the root of tree T and the list $\mathcal{D}_{p_u}^T$ has been computed for the parent p_u of u, then we define π_u^T to be the pointer to the node in list $\mathcal{D}_{p_u}^T$ that stores the vertex u. If u is the root of T, then π_u^T is null.

The extra data structures $\mathcal{D}_u^T$, $\mathcal{A}_u^T$, $\mathcal{Z}_u^T$, and π_u^T will always be used together with label $\lambda_{u,T}$. Hence, we give a special name to a tree that has all of these five pieces of information stored at each vertex. A rooted tree T is called *strongly labelled* if $\lambda_{u,T}$, $\mathcal{D}_u^T$, $\mathcal{A}_u^T$, $\mathcal{Z}_u^T$, and π_u^T are stored at each vertex u in T. The collection of all these five structures for a vertex u in T is called the *strong labelling of u*. The collection of strong labellings of all vertices in tree T is called the *strong labelling of T*. For reference, we now state the rather obvious fact that the strong labelling of a tree can be computed together with the vertex labelling of the tree without increasing the asymptotic running time.

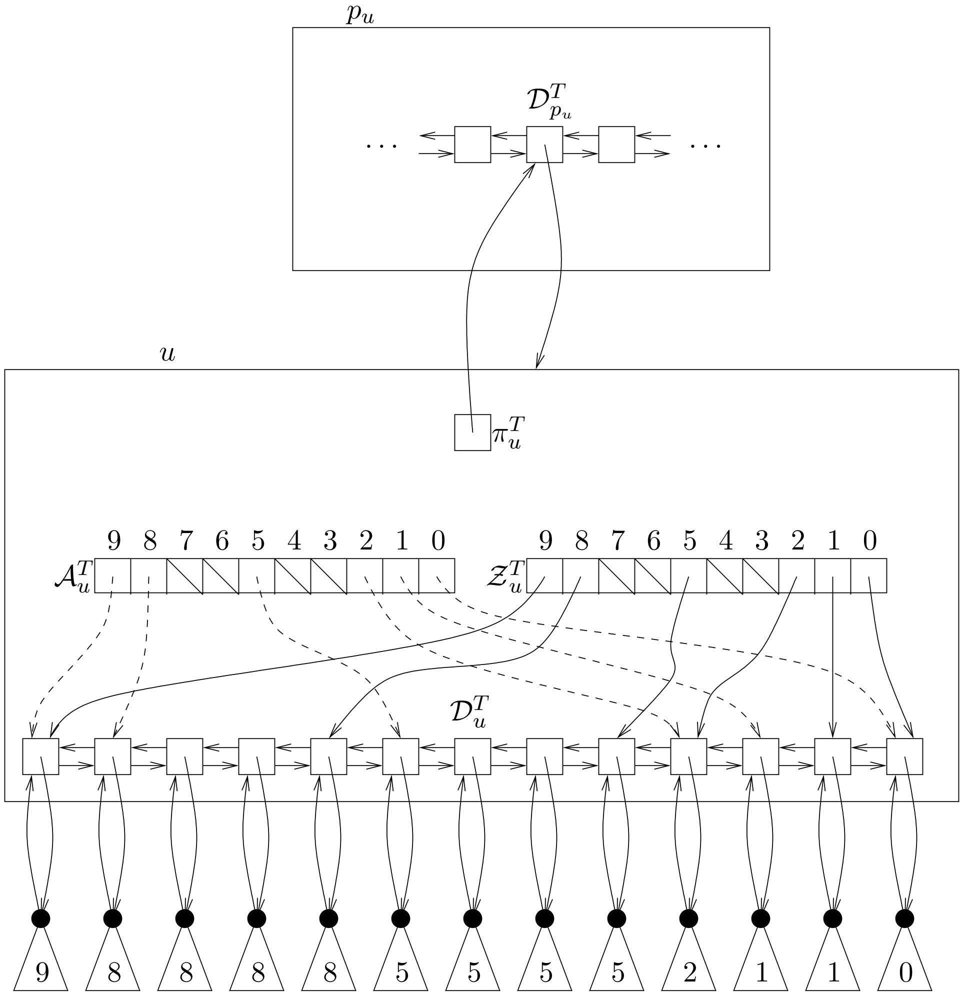

FIGURE 6.1: Extra information stored at each vertex u of a tree T. The vertex separations of the subtrees yielded by vertex u are indicated inside the triangles representing the subtrees.

Theorem 6.1 The strong labelling of a rooted tree T can be computed in time $O(|T|\text{VS}(T)) = O(|T| \lg |T|)$. $\square$

Hence, computing the strong labelling of a tree takes no more time than computing the vertex labelling of the tree using algorithm `COMBINE-LABELS` (Theorem 4.46).

Updating $\mathcal{D}^T_u$, $\mathcal{A}^T_u$, $\mathcal{Z}^T_u$, and π^T_u after a tree is attached to vertex u or a subtree yielded by u is removed is also simple, and we will not go into the details.

Lemma 6.2 Given a tree T and a vertex u in T, list $\mathcal{D}_u^T$, arrays $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$, and pointer π_u^T can be updated in $O(\text{VS}(T)) = O(\lg |T|)$ time after a subtree yielded by u is removed from T. The structures can be updated in $O(\text{VS}(T \vdash_u S)) = O(\lg |T \vdash_u S|)$ time after a tree S is attached to u. $\square$

6.2 Speeding Up Algorithm `COMBINE-LABELS`

Having described the list $\mathcal{D}_u^T$ and arrays $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$, we now show how to quickly combine the labels of the children of u using list $\mathcal{D}_u^T$. The key is to look only at $O(M)$ labels, where M is the largest vertex separation of a subtree yielded by u, even if there are $\Omega(|T|)$ children. Our algorithm is called `FAST-COMBINE-LABELS` and runs in time $O(M^2)$. It is obtained by modifying algorithm `COMBINE-LABELS` (Algorithm 4.1), which runs in time $\Theta(dM)$ (Lemma 4.45), where d is the number of children of vertex u.

A natural question to ask at this point is whether an alternative representation of the tree, one that imposes an upper bound on the number of children of each internal vertex, preserves the vertex separation. If such a representation is possible, then we can use algorithm `COMBINE-LABELS` to compute the combination of the labels of the children in sublinear time. Although we do not have a proof of the claim that there is no such representation, we show that one such common representation does not necessarily preserve the vertex separation of the tree. The *left-child right-sibling representation* of a rooted tree T is a binary tree *rep*(T) such that the left child of each internal vertex u in *rep*(T) is the leftmost child of u in T, and the right child of u in *rep*(T) is the right sibling of u in T. Figure 6.2 gives an example of this definition. In the figure, tree T has vertex separation 1 (Lemma 4.3), since the dotted line is a 1-backbone of T. Tree *rep*(T) has vertex separation greater than 1, however, because of Lemma 4.4 and the fact that vertex u has three branches in *rep*(T), each of which is path P_2 and hence has vertex separation 1 (Lemma 2.3). Thus, $\text{VS}(T) \neq \text{VS}(\textit{rep}(T))$.

The key idea for improving the running time of algorithm `COMBINE-LABELS` is to start iterating at a higher value of k in the loop on lines 4–14. For example, if $m \geq 1$ is an integer such that m is an element of three or more input labels $\mu_1, \ldots, \mu_d$,

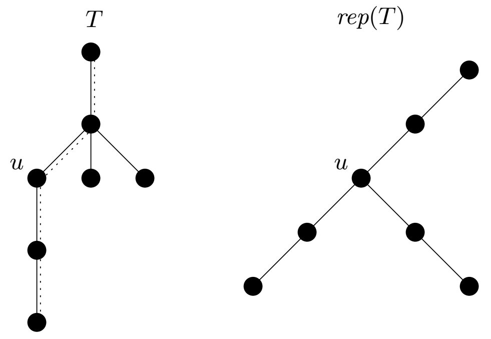

FIGURE 6.2: A rooted tree T and its left-child right-sibling representation $rep(T)$.

then λ is set to $(m+1)_0$ on line 6 of the algorithm during the iteration in which $k = m$; all smaller values of k are irrelevant in constructing the output label. This observation suggests that the loop starts iterating at m instead of 1. This idea does in fact work to reduce the running time of algorithm `COMBINE-LABELS`. However, it may be slow to check how many labels in $\mu_1, \ldots, \mu_d$ contain a particular integer. Instead, we focus only on the first elements of the labels, and denote by m the largest integer such that m is the first element of at least three labels in $\mu_1, \ldots, \mu_d$.

The modified algorithm, called `FAST-COMBINE-LABELS`, appears below. The new algorithm has only four lines that are different from those in algorithm `COMBINE-LABELS`; two new lines are added and two lines are modified.

The added lines appear before the first line of `COMBINE-LABELS`, and are numbered 0.1 and 0.2 in the pseudocode for `FAST-COMBINE-LABELS`. Line 0.1 finds the largest value of m such that m is the first element of at least three input labels, as discussed above, or sets m to 0 if no integer is the first element of at least three input labels. Line 0.2 then sets λ to $(m+1)_0$ if $m > 0$, which is consistent with the value of λ in algorithm `COMBINE-LABELS` at the end of the iteration in which $k = m$.

The lines of pseudocode for `FAST-COMBINE-LABELS` that are modifications of lines in algorithm `COMBINE-LABELS` are lines 1 and 4 and are primed in the pseudocode for `FAST-COMBINE-LABELS`. Line $1'$ differs from line 1 only in that the `if` statement has an `else` in front of it; this means that if $m > 0$, then the base case of algorithm `COMBINE-LABELS` on lines 1 and 2 is not executed. Line $4'$ differs from line 4 in the range of k over which the loop iterates; whereas COMBINE-LABELS starts the iteration at 1, FAST-COMBINE-LABELS starts at $m+1$. We now give the pseudocode for FAST-COMBINE-LABELS, and then prove its correctness.

Algorithm 6.1
Input: a sequence σ of vertex labels $\mu_1, \dots, \mu_d$.
Output: the combination λ of labels in σ.

```
FAST-COMBINE-LABELS($\mu_1, \dots, \mu_d$)
0.1.  $m \leftarrow$ the largest value of $m'$ such that $m'$ is the first element of at least
                three labels in $\sigma$; if $m'$ does not exist then $m \leftarrow 0$
0.2.  if $m > 0$ then $\lambda \leftarrow (m+1)_0$
1'.   else if 0 is an element of at least one label in $\sigma$ then $\lambda \leftarrow (1)_0$
2.    else $\lambda \leftarrow (0)_0$
3.    $M \leftarrow$ the maximum element of a label in $\sigma$ if $d > 0$, and 0 if $d = 0$
4'.   for $k$ from $m+1$ to $M$ do
5.        $i_k \leftarrow$ the number of labels in $\sigma$ that contain $k$
6.        if $i_k \geq 3$ then $\lambda \leftarrow (k+1)_0$
7.        if $i_k = 2$ then
8.            if $k$ is a critical element of at least one label in $\sigma$ then
                  $\lambda \leftarrow (k+1)_0$
9.            else $\lambda \leftarrow (k)_2$
10.       else if $i_k = 1$ then
11.           if $k$ is a critical element of the label in $\sigma$ that contains $k$ then
12.               if $\lambda(1) = k$ then $\lambda \leftarrow (k+1)_0$
13.               else $\lambda \leftarrow$ label $\lambda$ prepended with $k$
14.           else $\lambda \leftarrow (k)_1$
```

Lemma 6.3 Given a sequence σ of vertex labels $\mu_1, \dots, \mu_d$, Algorithm 6.1 on input $(\mu_1, \dots, \mu_d)$ correctly computes the combination λ of the labels in σ.

Proof. We prove the lemma by showing the analogue of Claim 4.44 for FAST-COMBINE-LABELS, which by induction immediately yields the correctness of the algorithm. Since the body of the loop is exactly the same in both algorithms COMBINE-LABELS and FAST-COMBINE-LABELS (lines 5–14), we only need to show that just before `FAST-COMBINE-LABELS` enters the loop on lines $4'$–14, or just after the loop terminates if $m = M$, label λ is equal to the combination of labels $\mu_1^m, \ldots, \mu_d^m$, since the loop starts iterating at $m+1$.

We now show the claim stated at the end of the last paragraph by analyzing the code of algorithm `FAST-COMBINE-LABELS` preceding the main loop. We consider two cases, depending on whether $m = 0$ or $m > 0$.

We first analyze the case when m, as set on line 0.1 of the algorithm, is 0. Then line $1'$ or 2 is executed and the loop starts iterating at 1. But lines $1'$ and 2 correspond exactly to lines 1 and 2 of algorithm `COMBINE-LABELS`, respectively, and hence algorithm `FAST-COMBINE-LABELS` behaves in exactly the same way as algorithm `COMBINE-LABELS` when $m = 0$. Therefore, Lemma 4.45 implies `FAST-COMBINE-LABELS` is correct.

We next consider the case when $m > 0$. Then m is the largest integer such that m is the first element of at least three labels among those in sequence $\mu_1, \ldots, \mu_d$. Hence, m is an element of at least three labels in $\mu_1, \ldots, \mu_d$. This means that during the iteration in algorithm `COMBINE-LABELS` in which $k = m$, variable λ is set to $(m+1)_0$ on line 6 of that algorithm. Thus, it follows from Claim 4.44 that label $(m+1)_0$ is the correct value of λ just before the iteration in which $k = m+1$, or just after the loop terminates if $m = M$. But algorithm `FAST-COMBINE-LABELS` sets λ to this value on line 0.2, and therefore λ is equal to the combination of labels $\mu_1^m, \ldots, \mu_d^m$ just before `FAST-COMBINE-LABELS` enters the loop on lines $4'$–14, or just after the loop terminates if $m = M$. $\square$

Having shown that algorithm `FAST-COMBINE-LABELS` is correct, we now prove its $O(M^2)$ running time.

Lemma 6.4 Given a sequence σ of labels $\mu_1, \ldots, \mu_d$ of the children of a vertex u in a tree T, Algorithm 6.1 on input $(\mu_1, \ldots, \mu_d)$ can be implemented to run in time $O(M^2)$, where M is the largest element of a label in σ, assuming the list $\mathcal{D}_u^T$ and labels of all the children of vertex u have been computed.

Proof. Before showing how to implement different parts of the algorithm to run in the stated time bound, we note that passing labels $\mu_1, \ldots, \mu_d$ explicitly as the input to the algorithm can require $\Omega(dM)$ time, since there are d labels and Lemma 4.25

implies each label can have length $\Omega(M)$. However, the labels are stored with the children of vertex u, and hence no passing of arguments is required.

We first show how to implement lines 0.1, 0.2, and $1'$–3 of algorithm `FAST-COMBINE-LABELS` to run in time $O(M)$. We implement line 0.1 as a simple list traversal: we start at the beginning of the list $\mathcal{D}_u^T$, and for each value m' of vertex separation starting at M and ending at 0, we count the number of subtrees yielded by u that have vertex separation m'. We stop as soon as we count three subtrees with vertex separation m'; value m is set to this value m' of vertex separation. Since the list $\mathcal{D}_u^T$ is ordered so that the vertex separation of the subtrees is monotonically decreasing, and because the first element of the label of a child v of u equals the vertex separation of T_v (Lemma 4.19), the value m thus computed is equal to the largest integer that is the first element of at least three labels in $\mu_1, \ldots, \mu_d$. Furthermore, we never count more than three nodes for each particular value of vertex separation, and hence the running time of the traversal is $O(M)$.

Line $1'$ can be implemented in $O(1)$ time by simply determining whether the last node of $\mathcal{D}_u^T$ contains a vertex that is the root of a subtree with vertex separation 0. Similarly, line 3 can be implemented in $O(1)$ time by determining the vertex separation of the subtree whose root is stored in the first node of $\mathcal{D}_u^T$. Lines 0.2 and 2 of algorithm `FAST-COMBINE-LABELS` can trivially be implemented to run in constant time. Hence, the running time of lines 0.1, 0.2, and $1'$–3 is $O(M)$.

We now analyze the running time of the loop on lines $4'$–14. Consider the sublist $\mathcal{D}_2$ of list $\mathcal{D}_u^T$ starting at the beginning of the m-block of $\mathcal{D}_u^T$ and ending at the end of $\mathcal{D}_u^T$. The crucial observation is that because $\mathcal{D}_u^T$ is sorted by the nonincreasing value of the vertex separation of the subtrees rooted at the vertices stored in the list, the vertex separation of the subtree T_v rooted at a vertex v stored in $\mathcal{D}_2$ is at most m. By Lemma 4.19, the vertex separation of T_v equals the first element of the label λ_v of v. As a consequence of Lemma 4.20, no element of λ_v is greater than m. It follows that no element of the label of a vertex stored in sublist $\mathcal{D}_2$ is greater than m. Therefore, for $k > m$, we do not need to examine on line 5 of the algorithm any labels of vertices stored in the sublist $\mathcal{D}_2$.

We now consider the sublist $\mathcal{D}_1$ of $\mathcal{D}_u^T$ starting at the beginning of $\mathcal{D}_u^T$ and ending at the node just in front of the first node in $\mathcal{D}_2$. By the definition of m

as the largest integer that is the first element of at least three labels in $\mu_1, \ldots, \mu_d$, we conclude from Lemma 4.19 that m is the largest integer such that m is the vertex separation of at least three subtrees yielded by vertex u. Hence, there are at most two vertices v_1 and v_2 stored in $\mathcal{D}_1$ such that subtrees T_{v_1} and T_{v_2} have vertex separation i, $m+1 \leq i \leq M$. We also observe that if m does not exist, then $\mathcal{D}_1 = \mathcal{D}_u^T$, and there are at most two vertices v_1 and v_2 stored in $\mathcal{D}_1$ such that subtrees T_{v_1} and T_{v_2} have vertex separation i, $0 \leq i \leq M$. Thus, the length of $\mathcal{D}_1$ is at most $2(M+1) = O(M)$. It follows from the conclusion of the previous paragraph that line 5 of algorithm `FAST-COMBINE-LABELS` can be implemented in time $O(M)$ per iteration by examining the $O(M)$ labels of the vertices in $\mathcal{D}_1$, and determining in constant time, as explained next, for each label μ_j whether k is an element of μ_j. Since μ_j is a strictly decreasing sequence (Lemma 4.20), we keep track in μ_j of the currently examined element as the loop progresses. Therefore, it is possible to determine in constant time if k is in μ_j.

It remains to show that lines 6–14 of algorithm `FAST-COMBINE-LABELS` can be implemented to run in $O(M)$ time per iteration. Lines 6, 7, 9, 10, 12, and 14 trivially run in $O(1)$ time. Checking whether k is a critical element of a label μ_j on lines 8 and 11 can be implemented to run in constant time as follows. If k is the last element of μ_j, then its criticality is equal to the criticality of μ_j, which can be read off directly from μ_j. If k is not the last element of μ_j, then Lemma 4.30 implies it is critical. Finally, line 13, which may involve copying λ, runs in $O(M)$ time, since the largest element of label λ is $O(M)$ and therefore Lemma 4.25 yields $|\lambda| = O(M)$.

We conclude that each iteration of the loop takes $O(M)$ time, and since the loop iterates at most M times, it follows that the running time of the loop, and therefore the entire algorithm, is $O(M^2)$. □

Having at our disposal a faster version of algorithm `COMBINE-LABELS`, we can finally give algorithms to update the strong labelling of a vertex u in a tree T after a subtree yielded by u is removed (algorithm `REMOVE-SUBTREE`), or after another tree is attached to u (algorithm `ATTACH-TREE`). If u is the root of T, then both algorithms update the strong labelling, and therefore the vertex separation, of tree T after it is modified. We first give the pseudocode for algorithm `REMOVE-SUBTREE`, and prove its correctness and running time afterward.

Algorithm 6.2
Input: an internal vertex u of a strongly labelled tree T and a child rm of u.
Output: updated label $\lambda_{u,T}$, list $\mathcal{D}_u^T$, arrays $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$, and pointer π_{rm}^T after the subtree T_{rm} is removed from T.

```
REMOVE-SUBTREE($u, rm$)
1.      update $\mathcal{D}_u^T$, $\mathcal{A}_u^T$, $\mathcal{Z}_u^T$, and $\pi_{rm}^T$
2.      $\lambda_u \leftarrow$ FAST-COMBINE-LABELS(the labels of the children of $u$ in $T\langle rm\rangle$)
```

Theorem 6.5 Given a strongly labelled tree T, an internal vertex u of T, and a child rm of u, Algorithm 6.2 on input (u, rm) correctly updates the label $\lambda_{u,T}$, list $\mathcal{D}_u^T$, arrays $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$, and pointer π_{rm}^T after the subtree T_{rm} is removed from T. The algorithm runs in time $O(\text{VS}(T)^2) = O\left(\lg^2 |T|\right)$.

Proof. The proof essentially combines results we have already established. Lemma 6.3 implies line 2 then correctly updates the label of vertex u in tree $T\langle rm\rangle$. The running times of lines 1 and 2 are $O(\text{VS}(T))$ (Lemma 6.2) and $O(M^2)$ (Lemma 6.4), respectively, where M is the largest element of a label "passed" as an argument on line 2. Since the largest element of the label of a vertex equals the vertex separation of the subtree rooted at that vertex (Lemma 4.19), and because every subtree yielded by vertex u is a subtree of tree T, Lemma 2.1 implies $M \leq \text{VS}(T)$. It follows that the running time of the algorithm is $O(\text{VS}(T)^2)$, which by Lemma 4.6 is $O\left(\lg^2 |T|\right)$. □

We now present the pseudocode for algorithm ATTACH-TREE, and prove its correctness and running time.

Algorithm 6.3
Input: a vertex u in a strongly labelled tree T and the root r_S of a strongly labelled tree S.
Output: updated label $\lambda_{u,T}$, list $\mathcal{D}_u^T$, arrays $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$, and pointer $\pi_{r_S}^S$ after tree S is attached to u via the edge ur_S.

```
ATTACH-TREE($u, r_S$)
```

```
1.      update $\mathcal{D}_u^T$, $\mathcal{A}_u^T$, $\mathcal{Z}_u^T$, and $\pi_{r_S}^S$
2.      $\lambda_u \leftarrow$ FAST-COMBINE-LABELS(the labels of the children of $u$ in $T \vdash_u S$)
```

Theorem 6.6 Given a strongly labelled tree T, a vertex u in T, and the root r_S of a strongly labelled tree S, Algorithm 6.3 on input (u, r_S) correctly updates the label $\lambda_{u,T}$, list $\mathcal{D}_u^T$, arrays $\mathcal{A}_u^T$ and $\mathcal{Z}_u^T$, and pointer $\pi_{r_S}^S$ after tree S is attached to T via the edge ur_S. The algorithm runs in time $O(\text{VS}(R)^2) = O\left(\lg^2 |R|\right)$, where $R = T \vdash_u S$.

Proof. The proof is similar to that of Theorem 6.5. Lemma 6.3 implies line 2 correctly updates the label of vertex u. It follows from Lemma 6.2 that the running time of line 1 is $O(\text{VS}(R))$. By Lemma 6.4, the running time of line 2 is $O(M^2)$, where M is the maximum element of a label passed as an argument on line 2; that is, M is the maximum vertex separation of a subtree yielded by u in tree R. It follows from Lemma 2.1 that $M \leq \text{VS}(R)$, and therefore Lemma 4.6 implies the running time of the algorithm is $O(\text{VS}(R)^2) = O\left(\lg^2 |R|\right)$. □

Algorithms `REMOVE-SUBTREE` and `ATTACH-TREE` update the strong labelling of tree T after it is modified if $u = r_T$. In the next chapter, we will see how to use algorithms `REMOVE-SUBTREE` and `ATTACH-TREE` to update the strong labelling of tree T after the tree is modified at an arbitrary vertex u.

Chapter 7

Modifying a Tree at an Arbitrary Vertex

This chapter generalizes the problems of Chapters 5 and 6. In Chapter 5, we presented an algorithm that computes the vertex labelling of tree $T \vdash S$ from the vertex labellings of trees T and S. The algorithm runs in time $O(\mathrm{VS}(T \vdash S))$. In Chapter 6, an alternative $O\left(\mathrm{VS}(T \vdash S)^2\right)$-time algorithm for the problem was given. That algorithm, in addition to updating the vertex labelling of $T \vdash S$, also updates the tree's strong labelling. We also gave an $O\left(\mathrm{VS}(T)^2\right)$-time algorithm for updating the strong labelling of a tree T after a subtree yielded by the root r_T of T is removed from the tree. In this chapter, we investigate the more general case when tree S is attached to an arbitrary vertex u in T, or a subtree of T yielded by an internal vertex u in T is removed from T. The algorithms are very similar to each other and achieve $O\left(m^{\log_3 2} \lg^2 m\right)$ amortized time complexity over a sequence of m constant-size tree additions to an initially constant-size tree, or sequence of m constant-size subtree removals from a tree such that the final tree has constant size.

Both algorithms work by traversing a subpath of the simple path $sp(u, r_T)$ from vertex u to the root r_T of tree T, updating the strong labelling of each vertex on that path. Since the length of $sp(u, r_T)$ can be linear in the number of vertices in tree T, traversing the path entirely each time a tree is attached to T or a subtree is removed from T does not necessarily yield sublinear amortized running time.

However, we observe that sometimes we do not need to traverse the entire path $sp(u, r_T)$; as soon as we encounter a vertex whose label does not change after the update, all ancestors of that vertex in $sp(u, r_T)$ do not need to be updated; this is quite obvious, but will be shown rigorously in Lemma 7.3.

Definition 7.1 The *red path* is the subpath from vertex u to vertex z of the simple path $sp(u, r_T)$, where z is the closest vertex to u in $sp(u, r_T)$ such that $\lambda_{z,T} = \lambda_{z,T \vdash_u S}$ if S is attached to T or $\lambda_{z,T} = \lambda_{z,T\langle v \rangle}$ if T_v is removed from T, where v is a child of u. If vertex z does not exist, then the red path is defined to be $sp(u, r_T)$.

We remark that the orientation of the red path is important; that is, the left endpoint of the red path is u and the right endpoint is z or r_T.

In the following claim, we collect a few observations that will be used in the proof that we only need to update the strong labelling of vertices on the red path. The claim follows immediately from Definition 7.1, and therefore a proof is omitted.

Claim 7.1 Given a tree T with root r_T, consider the red path in T after a tree S is attached to a vertex u in T or a subtree yielded by u is removed from T. Then

1. all vertices on the red path except possibly its right endpoint are such that their labels change after tree T is modified, and

2. the label of the right endpoint of the red path changes if and only if vertex z, as used in Definition 7.1, does not exist; in this case, the right endpoint of the red path is the root r_T. □

Shortly (Lemma 7.3), we will state and prove that the strong labelling of all vertices not on the red path remains "almost" the same after tree T is modified. Before doing that, however, we show a simple claim that will be useful in the proof.

Claim 7.2 Given a tree T with root r_T and an arbitrary vertex u in T, consider the red path in T after a tree S is attached to u or a subtree yielded by u is removed from T. If w is a vertex in either T or S not on the red path, the labels of no children of w change when tree T is modified.

Proof. To prove the claim, we consider two cases, depending on whether w is on the simple path $sp(u, r_T)$ or not.

We first consider the case when w is in $sp(u, r_T)$. We prove the case by induction on the length L_w of the simple path $sp(last_{RP}, w)$, where $last_{RP}$ is the right endpoint of the red path RP. We observe that $sp(u, r_T) = RP + sp\,(p, r_T)$, where p is the parent of vertex $last_{RP}$ in tree T. Because the assumption is that vertex w is on path $sp(u, r_T)$ but not on red path RP, it follows that w is in $sp\,(p, r_T)$. Thus, $L_w = |sp(last_{RP}, w)| = 1 + |sp\,(p, w)\,| \geq 1$, and one subtree R_{RP} yielded by w contains red path RP. We conclude that R_{RP} is the only subtree that becomes modified as a tree is attached to or a subtree is removed from tree T. Therefore, Definition 4.4 implies the labels of all other subtrees yielded by w remain the same. Hence, we only need to show that the label of tree R_{RP} does not change.

We first show the base case ($L_w = 1$) of the induction proof. In this case, $sp(last_{RP}, w) = last_{RP} + w$. This implies the right endpoint $last_{RP}$ of red path RP is a child of w. Since vertex w yields subtree R_{RP} and R_{RP} contains RP, we conclude that $last_{RP}$ is the root of R_{RP}. Because w is a descendant of root r_T, it follows that $r_T \neq last_{RP}$, and point 2 in Claim 7.1 implies the label of vertex $last_{RP}$ does not change; that is, the label of tree R_{RP} does not change.

We now show the induction step. We assume $L_w > 1$ and the labels of no children of vertex w_{RP} change, where w_{RP} is the child of w such that subtree $T_{w_{RP}}$ contains red path RP. This induction hypothesis is justified, since $|sp(last_{RP}, w)| = L_w$, and therefore $|sp(last_{RP}, w_{RP})| = L_w - 1 \geq 1$ (thus, tree $T_{w_{RP}}$ entirely contains RP). It follows that the label of vertex w_{RP} does not change, since by Lemma 4.34 the label of a vertex is a function of the labels of its children. As $T_{w_{RP}} = R_{RP}$, the label of subtree R_{RP} does not change.

We next consider the case when w is not on path $sp(u, r_T)$. In this case, none of the subtrees yielded by w change by attaching tree S to T or removing a subtree from T; if a subtree yielded by w became modified, then w would be on the path $sp(u, r_T)$. Since none of the subtrees yielded by w are modified, Definition 4.4 implies their labels remain unchanged. $\square$

We finally state and prove that the strong labelling of all vertices not on the red path remains the same after tree T is modified, except for one pointer π (Definition

6.2).

Lemma 7.3 Suppose that a strongly labelled tree S with root r_S is attached to a strongly labelled tree T with root r_T to form tree $T \vdash_u S$, where u is a vertex in T, or that subtree T_v is removed from T, where v is a child of u. Then the strong labelling of each vertex in either T or S not on the red path does not change as tree T is modified, except for $\pi^S_{r_S}$ if S is attached to T and π^T_v if T_v is removed from T.

Proof. In this proof, we consider a vertex w that is either in tree T or in tree S, and that is not on the red path. To simplify the argument, we define tree X as follows: if w is in T, then $X = T$, and if w is in S, then $X = S$. The proof is organized in the following way. First, we show that if $w \neq r_S$ and $w \neq v$, then the pointer π^X_w does not change. Second, we prove that the label $\lambda_{w,X}$, list $\mathcal{D}^X_w$, and arrays $\mathcal{A}^X_w$ and $\mathcal{Z}^X_w$ do not change.

First, we show that the pointer π^X_w does not change if $w \neq r_S$ and $w \neq v$. Suppose that $w \neq r_S$ and $w \neq v$. If $w = r_T$, then π^X_w is null before tree T is modified and is null after T is modified, because r_T is the root of both tree $T \vdash_u S$ and tree $T\langle v \rangle$. Hence, π^X_w does not change in this case. If $w \neq r_T$, then π^X_w points to the node in list $\mathcal{D}^X_{p_w}$ that stores vertex w, where p_w is the parent of w in tree X; we remark that w has a parent, because $w \neq r_S$ and $w \neq r_T$. Therefore, π^X_w does not change unless the edge between w and p_w is broken or created as the tree is modified. But the only vertex in either tree T or tree S for which the parent is different in tree $T \vdash_u S$ or T_v is r_S if S is attached to T and v if T_v is removed from T. Because $w \neq r_S$ and $w \neq v$ by the original supposition, we conclude that pointer π^X_w does not change as tree T is modified.

Second, we show that $\lambda_{w,X}$, $\mathcal{D}^X_w$, $\mathcal{A}^X_w$ and $\mathcal{Z}^X_w$ do not change. According to Lemma 4.34, the label of vertex w depends only on the labels of the children of w. By Claim 7.2, the labels of the children of w do not change, and therefore the label $\lambda_{w,X}$ of w does not change. Furthermore, since the labels of the subtrees yielded by w do not change, neither do their vertex separations (Lemma 4.19). Since the list $\mathcal{D}^X_w$ depends only on the vertex separations of the subtrees yielded by w, the list does not change. Finally, arrays $\mathcal{A}^X_w$ and $\mathcal{Z}^X_w$ depend only on $\mathcal{D}^X_w$. Because the list does not change, neither do the arrays. □

Lemma 7.3 implies we only need to update the strong labelling for vertices on the red path, and pointer $\pi^S_{r_S}$ if S is attached to T and pointer π^T_v if T_v is removed from T.

Before presenting the algorithms to update the strong labelling of tree T after attaching a tree to or removing a subtree from T, we first prove that the amortized length of the red path is $O\left(M(Mm)^{\log_3 2}\right)$ over a sequence of m tree additions or subtree removals; M is an upper bound on the size of each tree that is added or removed, and also on the sizes of the initial or final tree, depending on whether we are adding trees or removing subtrees, respectively. Since the algorithms traverse the red path and spend $O\left(\lg^2(Mm)\right)$ time for each vertex on the path (this will be shown later in Theorems 7.10 and 7.12), we will be able to conclude that the amortized running time of the algorithms is $O\left(M(Mm)^{\log_3 2}\lg^2(Mm)\right)$.

7.1 Binary Representation of a Vertex Label

In order to prove the bound on the amortized length of the red path, we need a new concept: a correspondence between vertex labels and binary numbers. We can think of a vertex label λ with maximum element $\lambda(1)$ as a binary number b_λ containing $\lambda(1)+1$ bits and defined as follows. If i is an element of λ, where $0 \leq i \leq \lambda(1)$, then b_λ has a 1 at position i; otherwise, b_λ contains a 0 at this position. For example, label $(5, 3, 1, 0)$ can be represented as 101011. It follows from Lemma 4.20 that this construction is a one-to-one correspondence up to the label's criticality. We call the binary number b_λ the *binary representation of label* λ. The following lemma gives a formula for computing the value of the binary representation of a label; it follows directly from the definition.

Lemma 7.4 The value of the binary representation of a label λ is $\sum_{j=1}^{|\lambda|} 2^{\lambda(j)}$. □

The next basic result will be useful in bounding the number of possible binary representations of a label in terms of the label's maximum element. It is another direct consequence of the definition and Lemma 4.20, but we state it explicitly for easy reference later.

Lemma 7.5 The binary representation of a label λ has at most $k+1$ bits, where k is an upper bound on $\lambda(1)$. $\square$

It will also be useful to refer to the uniqueness of the value of the binary representation of a label, which in conjunction with Lemma 7.5 makes it possible to count the number of possible labels with maximum element bounded above by k.

Lemma 7.6 The value of the binary representation of a vertex label is unique up to the label's criticality.

Proof. The lemma follows from Lemma 4.20 and observing that the value of a binary number b is unique once we specify the positions in b of 1's and 0's. $\square$

Our first goal is to show that when tree S is attached to tree T to form tree $T \vdash_u S$, where u is a vertex in T, the binary representation of the label λ of any vertex w in T does not decrease, and if it stays the same, then the criticality of λ does not decrease. This result will be important in bounding the number of times the label of a particular vertex can change, ultimately leading to the claimed amortized length of the red path.

Intuitively, decreasing the value of the binary representation of the label of a vertex corresponds to decreasing the vertex separation of a subtree (or subtrees) of the subtree rooted at that vertex. However, since tree T is a subtree of tree $T \vdash_u S$, every subtree of T is a subtree of $T \vdash_u S$, and hence by Lemma 2.1 the vertex separation of no subtree of T decreases. Therefore, the value of the binary representation of the label cannot decrease. The proof of this fact is technical, however, and we also need to take into account the label's criticality; it is deferred to Appendix A.

Lemma 7.7 If T and T' are trees with identical roots such that T is a subtree of T', and w is a vertex in T, then either $b_{\lambda'} > b_\lambda$, or $b_{\lambda'} = b_\lambda$ and $crit(\lambda') \geq crit(\lambda)$, where λ' and λ are the labels of w in trees T' and T, respectively.

Proof. See Appendix A. $\square$

Lemma 7.7 is the key result for bounding the number of times the label of a vertex in tree T can change as trees are attached to T. This bound in turn implies a bound on the sum of the lengths of the red paths, which can be used to bound the amortized length of the red path over a sequence of tree additions.

Lemma 7.8 Over a sequence of m tree additions to a tree, the amortized length of the red path is $O\left(M(Mm)^{\log_3 2}\right)$ for a single tree addition; M is an upper bound on the size of each tree added and the initial tree.

Proof. We begin the proof by deriving a relationship between the maximum vertex separation $\mathrm{VS}_{\mathrm{max}}$ of the tree at any point during the sequence of tree additions and the maximum number of vertices in the tree. We then use this result to bound the number of times a vertex label in the tree can change.

We first derive a relationship between $\mathrm{VS}_{\mathrm{max}}$ and the maximum number of vertices in the tree. By Lemma 4.5, the minimum number of vertices n_{min} in a tree with vertex separation $\mathrm{VS}_{\mathrm{max}}$ is

$$n_{\mathrm{min}} = \left(\frac{5}{6}\right) 3^{\mathrm{VS}_{\mathrm{max}}} - \frac{1}{2}. \tag{7.1}$$

Inverting Equation 7.1, we obtain

$$\mathrm{VS}_{\mathrm{max}} = \log_3 \frac{6n_{\mathrm{min}} + 3}{5}. \tag{7.2}$$

By supposition, each tree added and the initial tree have size at most M. Since the sequence consists of m tree additions, the number of vertices in the final tree is at most $M + Mm = M(1 + m)$. Denoting by T_i the tree resulting from performing the first i tree additions to the initial tree T_0, it is clear that tree T_i is a subtree of tree T_j if $i < j$. Lemma 2.1 therefore implies $\mathrm{VS}(T_i) \leq \mathrm{VS}(T_j)$, and hence $\mathrm{VS}_{\mathrm{max}} = \mathrm{VS}(T_m)$. By definition, the minimum number of vertices in a tree with vertex separation $\mathrm{VS}_{\mathrm{max}}$ is n_{min}, and because the number of vertices in the final tree T_m is at most $M(1 + m)$, we conclude that $n_{\mathrm{min}} \leq M(1 + m)$. Equation 7.2 now yields

$$\mathrm{VS}_{\mathrm{max}} \leq \log_3 \frac{6M(1 + m) + 3}{5}. \tag{7.3}$$

Having derived a relationship between the maximum vertex separation of and maximum number vertices in the tree over a sequence of m tree additions, we now use it to bound the number of times the label λ_w of any particular vertex w in the tree changes. Since the maximum vertex separation of the tree over the sequence of tree additions is $\mathrm{VS}_{\max}$, it follows from Lemma 2.1 that the subtree rooted at w has vertex separation at most $\mathrm{VS}_{\max}$ over the sequence; that is, $\lambda_w(1) \leq \mathrm{VS}_{\max}$. Hence, Lemma 7.5 implies the length of the binary representation of label λ_w is at most $\mathrm{VS}_{\max} + 1$. Because there are at most $2^{\mathrm{VS}_{\max}+1}$ possible binary numbers on at most $\mathrm{VS}_{\max} + 1$ bits, we conclude from Equation 7.3 that there are at most

$$2^{\mathrm{VS}_{\max}+1} \leq 2^{\log_3 \frac{6M(1+m)+3}{5}} = O\left((M(1+m))^{\log_3 2}\right) = O\left((Mm)^{\log_3 2}\right) \tag{7.4}$$

possible binary representations of label λ_w over the sequence of tree additions.

We now investigate how often the binary representation of λ_w can change. By Lemma 7.7 and the fact that tree T_i is a subtree of tree T_j for all $i < j$, it follows that the value of the binary representation b_j of label $\lambda_{w,j}$ in T_j is at least as large as the value of the binary representation b_i of $\lambda_{w,i}$ in T_i; that is, $b_j \geq b_i$. This implies by Equation 7.4 that the binary representation of λ_w changes $O\left((Mm)^{\log_3 2}\right)$ times over the course of the sequence of tree additions. By the uniqueness of the binary representation of a vertex label (Lemma 7.6), we infer that label λ_w changes $O\left((Mm)^{\log_3 2}\right)$ times up to criticality over the sequence.

If the binary representation of λ_w does not change by performing a tree addition (that is, if $b_{i+1} = b_i$), then it follows from Lemma 7.7 that $crit(\lambda_{w,i+1}) \geq crit(\lambda_{w,i})$. That is, if the label λ_w does not change up to criticality when a tree is attached to tree T_i to form T_{i+1}, then its criticality remains the same, in which case λ_w does not change, or its criticality increases from 0 to 1 or 0 to 2, or it increases from 1 to 2 (Lemma 4.7). We infer that when λ_w remains the same up to criticality, $crit(\lambda_w)$ changes at most two times. We conclude that label λ_w changes at most $2 \cdot O\left((Mm)^{\log_3 2}\right) = O\left((Mm)^{\log_3 2}\right)$ times over the sequence of m tree additions.

Finally, we derive the amortized length of the red path over the sequence of m tree additions. Since the label of a vertex in the dynamically changing tree chages $O\left((Mm)^{\log_3 2}\right)$ times and there are at most $M(1+m) = O(Mm)$ vertices in the tree, the maximum number of label changes in the tree is $O\left(Mm(Mm)^{\log_3 2}\right)$. The red path consists only of vertices whose labels change plus possibly one vertex whose label does not change (point 1 in Claim 7.1). Therefore, the sum of the lengths of the red paths is at most $m + O\left(Mm(Mm)^{\log_3 2}\right)$, since there are m tree modifications and therefore m red paths. We conclude that the amortized length of the red path for a tree addition is $\frac{m+O\left(Mm(Mm)^{\log_3 2}\right)}{m} = O\left(M(Mm)^{\log_3 2}\right)$. □

We observe that if M is a constant independent of m, the amortized length of the red path is $O\left(m^{\log_3 2}\right)$. Equipped with Lemma 7.8, we can easily prove the running time of the update algorithms described in the next section.

7.2 Update Algorithms

We now give a straightforward algorithm, called `ATTACH-TREE-ANYWHERE`, that traverses the red path RP of a tree T after another tree S is attached to T at vertex a, and updates the strong labelling of all vertices in RP. That is, it updates the vertex separation of tree $T \vdash_a S$ and prepares the tree for another tree addition.

The algorithm works by traversing the red path in tree T upward from the point of attachment a of tree S. The right endpoint $last_{RP}$ of path RP is detected either when the root r_T of tree T is reached, or when the newly computed label of a vertex on the path equals the old label of the vertex (Definition 7.1). The algorithm initially updates the strong labelling of vertex a by using subroutine `ATTACH-TREE` (Algorithm 6.3). Then, excluding the left endpoint a of RP, red path RP is traversed from the parent p_a of a to the right endpoint $last_{RP}$ of RP.

The strong labelling of each vertex w in this traversal is updated as follows. First, the subtree W yielded by w to which tree S is attached is removed, and the strong labelling of w is updated by subroutine `REMOVE-SUBTREE` (Algorithm 6.2). Second, the subtree $W \vdash_a S$ is attached to vertex w to form tree $T \vdash_a S$, and algorithm `ATTACH-TREE` is used to update the strong labelling of w. Since path RP is traversed upward, the strong labelling of the root r_W of W has already been updated when the algorithm is updating the strong labelling of w. The algorithm is summarized below.

Algorithm 7.1
Input: the roots r_T and r_S of strongly labelled trees T and S, respectively, and a vertex a in T.
Output: the strong labelling of tree $T \vdash_a S$.

```
ATTACH-TREE-ANYWHERE($r_T, r_S, a$)
1.    ATTACH-TREE($a, r_S$)
2.    if $a = r_T$ then exit
3.    for each vertex $w$ on the simple path from $p_a$ to $r_T$ in order do
4.        REMOVE-SUBTREE($w, r_W$)
5.        ATTACH-TREE($w, r_W$)
6.        if the new and old labels of $w$ are equal then exit
```

We now prove the correctness and amortized running time of the algorithm.

Theorem 7.9 Given strongly labelled trees T and S with roots r_T and r_S, respectively, and a vertex a in T, Algorithm 7.1 on input (r_T, r_S, a) correctly computes the strong labelling of tree $T \vdash_a S$.

Proof. The theorem follows quite easily from results of Chapter 6 and Lemma 7.3. Lemma 7.3 implies the strong labelling of no vertex in either tree T or tree S outside the red path changes as tree S is attached to tree T, except for the pointer $\pi^S_{r_S}$. But the strong labelling of each vertex on the red path is correctly updated on lines 1, 4, and 5 of the algorithm; this follows from Theorems 6.6 and 6.5. Theorem 6.6 also implies pointer $\pi^S_{r_S}$ is correctly updated on line 1. □

Theorem 7.10 Over a sequence of m tree additions to a tree, the amortized running time of Algorithm 7.1 is $O\left(M(Mm)^{\log_3 2}\lg^2(Mm)\right)$, where M is an upper bound on the size of each attached tree and the initial tree.

Proof. We prove the theorem by considering the running times of algorithms ATTACH-TREE and REMOVE-SUBTREE, and then the amortized length of the red path. We first derive the running time of one iteration of the algorithm, and then bound the number of iterations of the loop. Over the sequence of m tree additions, the maximum size of the tree is at most $M + Mm = O(Mm)$, since the initial tree and each added tree have size at most M. Theorem 6.6 therefore implies line 1 of algorithm ATTACH-TREE-ANYWHERE takes time $O\left(\lg^2(Mm)\right)$. Theorems 6.5 and 6.6 imply the running time of lines 4 and 5 for a single iteration of the loop on lines 3–6 is also $O\left(\lg^2(Mm)\right)$.

During each iteration, we also need to compare the newly computed label of vertex w with its old label in order to determine the end $last_{RP}$ of the red path RP (line 6). It follows from Theorem 4.6 and the fact that the number of vertices in the tree is $O(Mm)$ that the vertex separation of the tree is $O(\lg(Mm))$. Therefore, by Lemma 4.25 the lengths of the labels to be compared are both at most $O(\lg(Mm))$. Thus, we can compare the labels for equality in $O(\lg(Mm))$ time, including their criticalities; these are stored explicitly with each label. Because $O(\lg(Mm)) \leq O\left(\lg^2(Mm)\right)$, the running time of one iteration is $O\left(\lg^2(Mm)\right)$.

We now consider the amortized length of the red path RP, and use it to derive a bound on the amortized number of iterations of the loop on lines 3–6. By Lemma 7.8, the amortized length of RP is $O\left(M(Mm)^{\log_3 2}\right)$ over the sequence of tree additions. Hence, the amortized number of iterations of the loop is $O\left(M(Mm)^{\log_3 2}\right)$. We conclude that the amortized running time of the algorithm is $O\left(M(Mm)^{\log_3 2}\lg^2(Mm)\right)$, since the running time of one iteration was shown to be $O\left(\lg^2(Mm)\right)$ in the previous paragraph. $\square$

We next present an algorithm, called REMOVE-SUBTREE-ANYWHERE, for updating the strong labelling of tree T after a rooted subtree of T rooted at vertex rm is removed. It differs from algorithm ATTACH-TREE-ANYWHERE only in the code that precedes the loop; instead of calling subroutine ATTACH-TREE to update the strong labelling of vertex p_{rm}, algorithm REMOVE-SUBTREE is used to update the labelling. The traversal of the simple path from the parent $p_{p_{rm}}$ of vertex p_{rm} to the right endpoint $last_{RP}$ of the red path RP is exactly the same in both algorithms. For greater clarity and similarity to algorithm ATTACH-TREE-ANYWHERE, the arguments to algorithm REMOVE-SUBTREE-ANYWHERE include both the root rm of the removed subtree and parent p_{rm} of rm in tree T. This is redundant, because we can determine the parent of rm easily by reading a pointer. As in ATTACH-TREE-ANYWHERE, vertex r_W is the root of the subtree yielded by vertex w that has been modified.

Algorithm 7.2
Input: the root r_T of a strongly labelled tree T, a vertex p_{rm} in T, and a child rm of p_{rm} in T.
Output: the strong labelling of trees $T\langle rm\rangle$ and T_{rm}.

```
REMOVE-SUBTREE-ANYWHERE($r_T, p_{rm}, rm$)
1.    REMOVE-SUBTREE($p_{rm}, rm$)
2.    if $p_{rm} = r_T$ then exit
3.    for each vertex $w$ on the simple path from $p_{p_{rm}}$ to $r_T$ in order do
4.        REMOVE-SUBTREE($w, r_W$)
5.        ATTACH-TREE($w, r_W$)
6.        if the new and old labels of $w$ are equal then exit
```

We next prove the correctness of algorithm REMOVE-SUBTREE-ANYWHERE.

Theorem 7.11 Given a strongly labelled tree T with root r_T, and a vertex rm in T, Algorithm 7.2 on input (r_T, p_{rm}, rm) correctly computes the strong labelling of trees $T\langle rm\rangle$ and T_{rm}.

Proof. The proof follows a pattern similar to that of Theorem 7.9. By Lemma 7.3, the strong labelling of no vertex in tree T outside the red path changes as subtree T_{rm} is removed from tree T, except for the pointer π^T_{rm}. By Theorems 6.5 and 6.6, the strong labelling of each vertex on the red path is correctly updated on lines 1, 4, and 5 of the algorithm. Furthermore, it follows from Theorem 6.5 that pointer π^T_{rm} is correctly updated on line 1. $\square$

Finally, we show the amortized running time of algorithm REMOVE-SUBTREE-ANYWHERE.

Theorem 7.12 Over a sequence of m tree removals from a tree, the amortized running time of Algorithm 7.2 is $O\left(M(Mm)^{\log_3 2}\lg^2(Mm)\right)$, where M is an upper bound on the size of each removed subtree and the final tree.

Proof. Instead of duplicating the reasoning in the proof of Theorem 7.10, we prove this theorem by considering the corresponding sequence of tree additions. Denoting by σ the sequence of m subtree removals, we consider the sequence σ' that is the reverse of σ, and in which each tree is added rather than removed. Since the final tree in the subtree-removal sequence σ has size at most M, the initial tree in the tree-addition sequence σ has size at most M. Furthermore, every tree added in σ' has size at most M, because every subtree removed in σ has size at most M. Theorem 7.10 therefore implies the amortized running time of sequence σ' is $O\left(M(Mm)^{\log_3 2}\lg^2(Mm)\right)$. Clearly, the sequence of the strong labellings of the tree in sequence σ is the reverse of the sequence of the strong labellings of the tree in sequence σ', and therefore the same number of labels are updated in both sequences. We conclude that the amortized running time of algorithm `REMOVE-SUBTREE-ANYWHERE` is $O\left(M(Mm)^{\log_3 2}\lg^2(Mm)\right)$. $\square$

If the bound M is a constant independent of m, algorithms `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANYWHERE` have $O\left(m^{\log_3 2}\lg^2 m\right)$ amortized running time, which is sublinear in m, since $\log_3 2 \approx 0.63 < 1$. In a sequence of m tree additions or subtree removals, the tree potentially has $n = \Omega(m)$ vertices. Therefore, applying our algorithms to recompute the vertex separation of a dynamically changing tree as constant-size subtrees are attached to or removed from the tree is faster than recomputing the vertex separation of the tree by either the Ellis-Sudborough-Turner algorithm [EST94] or Skodinis' algorithm [Sko00]; both algorithms have running time $\Theta(n) = \Omega(m)$, where n is the number of vertices in the tree.

As a concluding thought, we remark that if both tree additions to and subtree removals from a tree are allowed in the sequence, the amortized red path length for a single operation can be $\Omega(Mm)$. As an example demonstrating the validity of this claim, consider the following sequence of operations. The first $\frac{m}{2}$ operations consist of adding trees of size M such that the resulting tree has size $\frac{Mm}{2}$ and depth $\Omega(Mm)$. The next $\frac{m}{2}$ operations of the sequence consist of repeatedly attaching and removing a subtree yielded by a vertex w at depth $\Omega(Mm)$ that causes the vertex separation of the tree to repeatedly increase and decrease by 1. Thus, the red path after each operation is the simple path from w to the root of the tree and has length $\Omega(Mm)$. Hence, the total cost of the second half of the sequence is $\frac{m}{2}\Omega(Mm) = \Omega(Mm^2)$, which implies that the amortized cost for an operation is $\frac{\Omega(Mm^2)}{m} = \Omega(Mm)$.

Chapter 8

Conclusion and Open Problems

In this thesis, we primarily looked at how to update the vertex separation of a tree T after another tree S is attached to T or after a rooted subtree is removed from T. We first described an $O(\lg |T \vdash S|)$-time algorithm, `ADD-LABEL`, that solves the problem by updating the vertex labelling of the tree if T and S are attached at their roots. The algorithm is essentially a case analysis of the labels of trees T and S.

Algorithm `ADD-LABEL` is used in our next algorithm, `LABEL`, to compute the vertex labelling, and therefore the vertex separation, of tree T. Although algorithm `LABEL` achieves the same running time, $O(|T| \lg |T|)$, as the Ellis-Sudborough-Turner algorithm to compute the vertex labelling of tree T, it uses a different technique. The technique is to split the input tree T into two parts, compute the vertex labellings of the two parts recursively, and then combine the vertex labellings using algorithm `ADD-LABEL`. We believe this technique is conceptually simpler than the technique employed by algorithm `COMBINE-LABELS`, which uses recursion on the children of each vertex in tree T.

One feature of algorithm `LABEL` worth discussing is that when splitting the tree T into trees $T\langle u\rangle$ and T_u, where u is a child of the root r_T of T, the algorithm chooses an arbitrary child u. It is conceivable that some children are better than others for the running time of the algorithm. We remark that the asymptotic worst-case running time of algorithm `LABEL` cannot be improved by making a more clever choice of u, since in the case when the input tree T is perfect, all subtrees yielded by any particular vertex in T are identical. However, it might be possible to optimize the choice of u for a particular input T, which is not necessarily as pathological as a perfect tree. The following example provides motivation for investigating this further.

Consider the case when child u of r_T is chosen so that $\mathrm{VS}(T\langle u\rangle) > \mathrm{VS}(T_u)$ and the canonical backbone $B_{T\langle u\rangle}$ of tree $T\langle u\rangle$ includes root r_T. Then tree T_u is a branch of $B_{T\langle u\rangle}$, and the label of r_T is the same regardless of the label of T_u; this can be proved using the results derived in this thesis. Thus, after computing the vertex labelling of tree $T\langle u\rangle$ recursively, if we can somehow deduce that the vertex separation of tree T_u is less than $\mathrm{VS}(T\langle u\rangle)$, we do not even need to apply recursion on tree T_u or find its vertex separation exactly. One way to tell that the vertex separation of T_u is too small is if the tree's size is too small. Even if T_u contains many vertices, its vertex separation is small if the tree is too skinny; that is, if it is essentially a long path with many small branches attached to it. If we can somehow quickly test these tree characteristics, we might be able to reduce the number of recursive calls.

All algorithms to date for computing the vertex separation of tree T require the tree to be rooted, and they examine all vertices in T. The fact that T needs to be rooted at an arbitrary vertex immediately raises the question of whether some choices for root r_T are better than others. In order to compare different choices for root r_T, we need to have a metric that establishes the "goodness" of r_T as the root of tree T. One such natural measure is the length of the concatenation of the labels of all vertices in T. The best possible scenario is when all labels have unit length; then, the vertex labelling of T can be computed in time linear in $|T|$. The fact that all algorithms examine all vertices in tree T implies their running time is $\Omega(|T|)$. An interesting open problem is whether we can quickly approximate a subtree T' of T such that $\mathrm{VS}(T') = \mathrm{VS}(T)$. If T' is small compared to T, then we can use Skodinis' algorithm to compute the vertex separation of tree T in $O(|T'|) = o(|T|)$ time.

In order to be able to update the vertex labelling of tree T after removing a subtree from T, and also to generalize the problem of updating the labelling after a tree is attached, we introduced the concept of strong labelling. The strong labelling of tree T essentially orders the subtrees yielded by each vertex in T by their vertex separations. The strong labelling also contains the vertex labelling of T, and it includes additional data structures that make it possible to quickly update the subtree ordering of a vertex after another tree is attached to the vertex, or after a subtree yielded by the vertex is removed. This leads to two algorithms, `REMOVE-SUBTREE` and `ATTACH-TREE`, that update the strong labelling of tree T with root r_T after a subtree of T yielded by r_T is removed from T, and after a strongly labelled tree S with root r_S is attached to T via the edge $r_T r_S$.

Algorithms `REMOVE-SUBTREE` and `ATTACH-TREE` run in times $O\left(\lg^2 |T|\right)$ and $O\left(\lg^2 |T \vdash S|\right)$, respectively. Algorithm `ADD-LABEL` is faster than `ATTACH-TREE` by a factor of $\lg |T|$, and it also updates the vertex separation of tree $T \vdash S$. However, `ATTACH-TREE` updates the strong labelling as well, which we need in the more general algorithms `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANYWHERE`. It is possible to combine the ideas behind algorithms `ADD-LABEL` and `ATTACH-TREE` so that the resulting algorithm updates the strong labelling of $T \vdash S$ and runs in time $O(\lg |T \vdash S|)$. We chose not to elaborate on this possibility, because algorithms `REMOVE-SUBTREE` and `ATTACH-TREE` are always used together as a pair, and the running time of `REMOVE-SUBTREE` is $O\left(\lg^2 |T|\right)$, thus dominating the running time of this potentially faster `ATTACH-TREE` algorithm.

It would be interesting to investigate a simpler scheme for keeping sufficient information with a tree to update its vertex separation as other trees are added to or removed from the tree. We use vertex and strong labellings, but it is conceivable that they contain redundant information. A simpler approach could greatly simplify the algorithms and potentially improve their running times as well. Our need to compute and update the strong labelling of a tree stems from having to be able to update the label of a vertex. Should a scheme be devised that does not use vertex labelling to update the vertex separation of a dynamically changing tree, there may be no need for strong labelling.

Finally, we gave algorithms, `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANY-WHERE`, that update the strong labelling of tree T after another strongly labelled tree S is attached to an arbitrary vertex u in T, and after a subtree yielded by a vertex u is removed from T. Although updating the strong labelling using these algorithms can be slow for a single tree modification, we showed that the amortized time complexity of the algorithms is sublinear in the number of modifications under the following conditions: either tree additions or subtree removals are allowed, but not both. Furthermore, for a sequence of tree additions, the initial tree and each tree added have size constant and independent of the number of tree additions. Similarly, for a sequence of subtree removals, each subtree removed and the final tree have size constant and independent of the number of subtree removals.

If the conditions described in the last paragraph are satisfied, then the amortized running time of each algorithm `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANYWHERE` is $O\left(m^{\log_3 2}\lg^2 m\right)$, where m is the number of tree additions or subtree removals. The key concept in deriving this time bound is that of the red path, which after each tree modification consists of the vertices whose strong labellings need to be updated, and whose amortized length was shown to be $O\left(m^{\log_3 2}\right)$.

We mentioned that if both tree additions and subtree removals are allowed in the sequence of tree modifications, the amortized length of the red path can be linear in the number of operations. Therefore, the amortized running time of algorithms `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANYWHERE` can be linear in the number of operations m. Finding an algorithm that updates the vertex separation of a tree in this case, and whose running time is sublinear in m, is an open problem. Such an algorithm would have to use an approach different from updating the vertex labelling of the tree, since we showed a linear lower bound on the number of vertex labels that can change after each tree modification.

Another point to consider is that the amortized length of the red path was derived using a global counting argument by bounding the number of labels in the tree that can change over an entire sequnce of tree additions. Our upper bound on the number of labels was essentially argued by the following reasoning: first, we argued that when a tree is added to tree T, the binary representation of the label of any vertex in T cannot decrease. Second, we derived an upper bound on the number of possible labels with maximum element at most k, and concluded that the number of possible labels is an upper bound on the number of times a label changes over the sequence of tree additions. Finally, we bounded k in terms of the number of tree additions.

Although this reasoning produced a sublinear bound on the amortized length of the red path, which was our objective, intuitively we expect the amortized length of the red path to be much smaller. In other words, we expect that bounding the number of labels occurring in a tree by the number of possible labels is not tight. This suggests that our upper bound on the length of the red path can be reduced significantly. We expect it to be possible to reduce the bound to a polylogarithmic one. Our reason for this conjecture is that because of the relationship between the minimum number of vertices in a tree with a particular vertex separation, we need to at least triple the size of the tree in order to increase its vertex separation by 1. In conclusion, it is an interesting open problem to reduce the current upper bound of $O\left(m^{\log_3 2}\right)$ on the amortized length of the red path. Finding pathological sequences of tree additions that cause the amortized length to be large is another avenue of research, and it can possibly provide more understanding of the relationship between the structure of a tree and the tree's vertex separation.

In addition to reducing the upper bound on the length of the red path by a tighter analysis, there may also be a way of reducing the $\lg^2 m$ factor in the running time of algorithms `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANYWHERE`. This can potentially be achieved by a more refined analysis of the label being updated; there are many cases when the label changes only very slightly. Another factor contributing to relatively fast average update times is that many labels in a typical tree have very short length. It would be interesting to investigate this a little bit further, and perhaps derive an upper bound on the length of the concatenation of all labels in a tree, or the amortized length of the concatentation of the labels on the red path.

The only tree modifications that we have considered are attaching another (rooted) tree and removing a (rooted) subtree. Expanding an edge involves replacing the edge by two edges and one new vertex, and contracting an edge involves removing a degree-2 vertex and connecting the two adjacent vertices by a new edge. Edge expansion can be viewed as removing a subtree, attaching a single vertex, and then reattaching the subtree. Similarly, edge contraction can be viewed as removing a subtree, removing a single vertex, and then reattaching the subtree. Therefore, the vertex labelling of the tree after both modifications can be updated in sublinear time using algorithms `ATTACH-TREE-ANYWHERE` and `REMOVE-SUBTREE-ANYWHERE`. However, since the modifications are minor, it might be possible to devise algorithms with much lower running time bounds.

Another tree "modification" that might be of interest is re-rooting the tree. In this case, it is not clear whether a sublinear-time algorithm to update the vertex labelling is possible. Since re-rooting a tree can completely change the ancestor-descendant relationship of all vertices, and because vertex labelling depends on this relationship, we conjecture that a sublinear-time algorithm for this problem is not possible. However, such an algorithm might be possible if the distance of the new root from the old root is bounded above by a constant, since then the ancestor-descendant relationship is partially preserved.

The next major step in the research presented in this thesis would be to devise algorithms that update an optimal layout of tree T with respect to vertex separation as trees are added to or removed from T. Storing an integer with each vertex in T to indicate its position in an optimal layout φ is probably not a good idea, since inserting a single vertex to T could potentially shift $\Omega(|T|)$ positions in φ. However, the amortized running time of an algorithm using this simple data structure might be good. Another, more reasonable approach is to encode optimal layout φ in a more clever way that would make both queries on φ and updates of tree T run quickly, preferably in polylogarithmic time. Devising such a data structure is certainly an exciting open problem.

A natural extension of our work to a related problem would be to updating the cutwidth of tree T after a tree is attached to T or a subtree is removed from T. Yannakakis' algorithm [Yan85] for computing the cutwidth of tree T in $O(|T| \lg |T|)$ time also uses the concept of vertex labelling, although the vertex labels are defined differently. It might be possible to use this different kind of vertex labelling to derive algorithms similar to those presented in this thesis, at least conceptually.

Appendix A

Proof of Lemma 7.7

We present the proof in stages by means of several claims. In the end, we will have shown that either $b_{\lambda'} > b_\lambda$, or $b_{\lambda'} = b_\lambda$ and $\mathit{crit}(\lambda') \geq \mathit{crit}(\lambda)$, where λ is the label of a vertex w in a tree T, λ' is the label of w in a tree T', and T is a subtree of T' such that the roots of trees T and T' are identical (subscript w on λ and λ' is omitted for clarity).

It is convenient to define additional notation for the purposes of the claims and their proofs. Additional notation is not strictly necessary, but it will greatly reduce clutter. In all claims we consider an arbitrary vertex w in tree T. Since T is a subtree of tree T', w is also in T'. The labels of w in trees T and T' are denoted by λ and λ', respectively. The vertices in tree T_w corresponding to elements $\lambda(1), \ldots, \lambda(|\lambda|)$ of label λ are denoted by $v_1, \ldots, v_{|\lambda|}$, respectively. Since we will need to reason about it, we denote this sequence of vertices by σ. Similarly, the vertices in tree T'_w corresponding to elements $\lambda'(1), \ldots, \lambda'(|\lambda'|)$ of label λ' are denoted by $v'_1, \ldots, v'_{|\lambda'|}$, respectively, and the sequence itself is denoted by σ'.

The proofs involve reasoning about the relationship between the sequences σ and σ'. The most important property of the two sequences is the length of their longest common prefix. We therefore give this length a special name: pf. We remark that $0 \leq pf \leq \min\{|\lambda|, |\lambda'|\}$ holds. Finally, we define $T_w\langle j\rangle = T_w\langle v_1, \ldots, v_j\rangle$ and $T'_w\langle j\rangle = T'_w\langle v'_1, \ldots, v'_j\rangle$.

We begin by proving an important relationship, used several times in subsequent proofs, between the vertex separations of the trees $T_w\langle j\rangle$ and $T'_w\langle j\rangle$.

Claim A.1 The following inequality holds for all j such that $1 \leq j \leq pf$:

$$\lambda'(j) = \mathrm{VS}\left(T'_w \langle j-1\rangle\right) \geq \mathrm{VS}\left(T_w \langle j-1\rangle\right) = \lambda(j).$$

Proof. The claim is an easy consequence of Lemma 2.7. If $j = 1$, then Lemmas 2.1 and 4.19 combined with the fact that tree T is a subtree of tree T' imply $\lambda'(1) = \mathrm{VS}(T'_w\langle\rangle) = \mathrm{VS}(T'_w) \geq \mathrm{VS}(T_w) = \mathrm{VS}(T_w\langle\rangle) = \lambda(1)$. When $2 \leq j \leq pf$, since $v_1 = v'_1, v_2 = v'_2, \ldots, v_{pf} = v'_{pf}$ by the definition of pf, Lemma 2.7 implies tree $T_w \langle j\rangle$ is a subtree of tree $T'_w \langle j\rangle$. Lemma 2.1 and Lemma 4.26 hence yield $\lambda'(j) = \mathrm{VS}\left(T'_w \langle j-1\rangle\right) \geq \mathrm{VS}(T_w \langle j-1\rangle) = \lambda(j)$. □

The next four claims are sufficient to easily show our goal that either $b_{\lambda'} > b_\lambda$, or $b_{\lambda'} = b_\lambda$ and $crit(\lambda') \geq crit(\lambda)$. First, we show that sequence σ cannot be a proper prefix of sequence σ' and vice versa; that is, the longest common prefix of sequences σ and σ' cannot be equal to one of the sequences and not equal to the other sequence.

Claim A.2 Sequence σ is not a proper prefix of sequence σ' and σ' is not a proper prefix of σ.

Proof. We show the claim by contradiction. Assume σ is a proper prefix of sequence σ' or σ' is a proper prefix of σ. We will derive a contradiction to Lemma 4.27. We only consider the case when σ is a proper prefix of sequence σ'; the other case is completely analogous. In this case, $v_{|\lambda|} = v'_{|\lambda|}$. Because $v_{|\lambda|}$ is the vertex corresponding to the last element $\lambda(|\lambda|)$ of label λ, Lemma 4.27 implies $v_{|\lambda|} = w$. Thus, $v'_{|\lambda|} = w$. But $v'_{|\lambda|}$ is not the last vertex in sequence σ', since σ is a proper prefix of σ' and hence $|\lambda'| > |\lambda|$. This contradicts Lemma 4.27, because λ' is the label of vertex w in tree T'. □

In the next claim, we consider the case when the longest common prefix of sequences σ and σ' is a proper prefix of both σ and σ', and vertex v_{pf+1} is critical in tree $T_w \langle pf\rangle$. The case when v_{pf+1} is noncritical in $T_w \langle pf\rangle$ will be considered immediately afterward.

Claim A.3 If the longest common prefix of sequences σ and σ' is a proper prefix of both σ and σ' and vertex v_{pf+1} is critical in tree $T_w \langle pf \rangle$, then $b_{\lambda'} > b_{\lambda}$.

Proof. The proof of this claim involves four steps. Initially, we prove that vertex v_{pf+1} is noncritical in tree $T'_w \langle pf \rangle$. Then, we prove that v_{pf+1} yields two subtrees in tree $T'_w \langle pf \rangle$ that have vertex separation at least $\lambda(pf+1)$. Next, we show how this implies $\lambda'(pf+1) > \lambda(pf+1)$, and finally argue that $b_{\lambda'} > b_{\lambda}$.

We first show that vertex v_{pf+1} is not critical in tree $T'_w \langle pf \rangle$. Since the sequence of vertices $v_1, ..., v_{pf}$ is the longest common prefix of sequences σ and σ', it follows that $v'_{pf+1} \neq v_{pf+1}$; otherwise, the longest common prefix could be extended, contradicting the fact that it is longest. One of the assumptions is that vertex v_{pf+1} is critical in tree $T_w \langle pf \rangle$. If v_{pf+1} is critical in tree $T'_w \langle pf \rangle$ as well, then Definition 4.4 implies $v_{pf+1} = v'_{pf+1}$ by the uniqueness of the critical vertex in a tree (Lemma 4.7). Thus, v_{pf+1} is not critical in $T'_w \langle pf \rangle$.

We next prove v_{pf+1} yields two subtrees in tree $T'_w \langle pf \rangle$ that have vertex separation at least $\lambda(pf+1)$. It follows from Lemma 2.7 combined with the fact $v_1 = v'_1, v_2 = v'_2, \ldots, v_{pf} = v'_{pf}$ that tree $T_w \langle pf \rangle$ is a subtree of tree $T'_w \langle pf \rangle$. Therefore, the subtree $T_w \langle pf \rangle_{v_{pf+1}}$ of $T_w \langle pf \rangle$ rooted at v_{pf+1} is a subtree of $T'_w \langle pf \rangle$, and hence it is a subtree of $T'_w \langle pf \rangle_{v_{pf+1}}$. Since v_{pf+1} is critical in $T_w \langle pf \rangle$, it yields two subtrees R_1 and R_2 in $T_w \langle pf \rangle$ with vertex separation $\mathrm{VS}\,(T_w \langle pf \rangle)$. Lemma 4.26 implies

$$\lambda(pf+1) = \mathrm{VS}(T_w \langle pf \rangle) = \mathrm{VS}(R_1) = \mathrm{VS}(R_2). \tag{A.1}$$

Since tree $T_w \langle pf \rangle_{v_{pf+1}}$ is a subtree of $T'_w \langle pf \rangle_{v_{pf+1}}$, vertex v_{pf+1} yields two subtrees R'_1 and R'_2 in $T'_w \langle pf \rangle$ such that R_1 is a subtree of R'_1 and R_2 is a subtree of R'_2. It follows from Lemma 2.1 that $\mathrm{VS}(R'_1) \geq \mathrm{VS}(R_1)$ and $\mathrm{VS}(R'_2) \geq \mathrm{VS}(R_2)$. In other words, vertex v_{pf+1} yields two subtrees, R'_1 and R'_2, in $T'_w \langle pf \rangle$ with vertex separation at least $\lambda(pf+1)$.

We now argue how the conclusions of the previous two paragraphs imply $\lambda'(pf+1) > \lambda(pf+1)$. Since tree $T'_w \langle pf \rangle$ has a subtree with vertex separation at least $\lambda(pf+1)$, Lemma 2.1 implies that $\mathrm{VS}\,(T'_w \langle pf \rangle) \geq \lambda(pf+1)$. If $\mathrm{VS}\,(T'_w \langle pf \rangle) = \lambda(pf+1)$, then it follows from Equation A.1 that $\mathrm{VS}\,(T'_w \langle pf \rangle) = \mathrm{VS}(R_1) = \mathrm{VS}(R_2)$. Since $\mathrm{VS}(R'_1) \geq \mathrm{VS}(R_1)$ and $\mathrm{VS}(R'_2) \geq \mathrm{VS}(R_2)$, Lemma 2.1 yields $\mathrm{VS}\,(T'_w \langle pf \rangle) = \mathrm{VS}(R'_1) = \mathrm{VS}(R'_2)$. We conclude that vertex v_{pf+1} is critical in $T'_w \langle pf \rangle$. But we argued that v_{pf+1} is not critical in tree $T'_w\langle pf\rangle$. Therefore, the vertex separation of $T'_w\langle pf\rangle$, which is $\lambda'(pf+1)$ (Lemma 4.26), is greater than $\lambda(pf+1)$; that is, $\lambda'(pf+1) > \lambda(pf+1)$.

Finally, we show that $b_{\lambda'} > b_\lambda$ holds by using the fact $\lambda'(pf+1) > \lambda(pf+1)$ proved in the previous paragraph. Lemma 7.4 implies the value of the binary representation b_λ is $\sum_{j=1}^{|\lambda|} 2^{\lambda(j)} = \sum_{j=1}^{pf} 2^{\lambda(j)} + 2^{\lambda(pf+1)} + \sum_{j=pf+2}^{|\lambda|} 2^{\lambda(j)}$. We bound from above the value of the term $\sum_{j=pf+2}^{|\lambda|} 2^{\lambda(j)}$ in order to prove that $b_{\lambda'} > b_\lambda$. Since a vertex label is a strictly decreasing sequence (Lemma 4.20), $\lambda(j-1) < \lambda(j) < \lambda(pf+1)$ holds for all j such that $j > pf+1$. It also follows that the length of the suffix $\lambda(pf+2), \ldots, \lambda(|\lambda|)$ of label λ is at most $\lambda(pf+1)$; this maximum length is achieved when the suffix is $\lambda(pf+1)-1, \lambda(pf+1)-2, \ldots, 1, 0$. Therefore,

$$\sum_{j=pf+2}^{|\lambda|} 2^{\lambda(j)} \leq \sum_{j=0}^{\lambda(pf+1)-1} 2^j = 2^{\lambda(pf+1)} - 1. \tag{A.2}$$

Having bounded from above the term $\sum_{j=pf+2}^{|\lambda|} 2^{\lambda(j)}$, we can now prove that $b_{\lambda'} > b_\lambda$:

$$\begin{aligned}
b_{\lambda'} &= \sum_{j=1}^{pf} 2^{\lambda'(j)} + 2^{\lambda'(pf+1)} + \sum_{j=pf+2}^{|\lambda'|} 2^{\lambda'(j)} && \text{(A.3)}\\
&\geq \sum_{j=1}^{pf} 2^{\lambda'(j)} + 2^{\lambda'(pf+1)} && \text{(A.4)}\\
&\geq \sum_{j=1}^{pf} 2^{\lambda(j)} + 2^{\lambda'(pf+1)} && \text{(A.5)}\\
&= \sum_{j=1}^{pf} 2^{\lambda(j)} + 2^{\lambda'(pf+1)-1} + 2^{\lambda'(pf+1)-1} && \text{(A.6)}\\
&\geq \sum_{j=1}^{pf} 2^{\lambda(j)} + 2^{\lambda(pf+1)} + 2^{\lambda(pf+1)} && \text{(A.7)}\\
&> \sum_{j=1}^{pf} 2^{\lambda(j)} + 2^{\lambda(pf+1)} + 2^{\lambda(pf+1)} - 1 && \text{(A.8)}
\end{aligned}$$

$$\geq \sum_{j=1}^{pf} 2^{\lambda(j)} + 2^{\lambda(pf+1)} + \sum_{j=pf+2}^{|\lambda|} 2^{\lambda(j)} \tag{A.9}$$

$$= b_\lambda; \tag{A.10}$$

Equation A.5 follows from Claim A.1, Equation A.7 follows from the fact, proved above, that $\lambda'(pf+1) > \lambda(pf+1)$, Equation A.9 is a consequence of Equation A.2, and Equation A.10 follows from Lemma 7.4. □

We now consider the case analyzed in Claim A.3, except that vertex v_{pf+1} is noncritical in tree $T_w\langle pf\rangle$. The conclusion reached is the same, but the proof technique is different.

Claim A.4 If the longest common prefix of sequences σ and σ' is a proper prefix of both σ and σ' and vertex v_{pf+1} is noncritical in tree $T_w\langle pf\rangle$, then $b_{\lambda'} > b_\lambda$.

Proof. We prove the claim by demonstrating that $|\lambda'| > |\lambda|$. Since vertex v_{pf+1} is noncritical in tree $T_w\langle pf\rangle$, it follows from Lemma 4.28 that $v_{pf+1} = w$. Therefore, Lemma 4.27 implies v_{pf+1} is the vertex corresponding to the last element of label λ; that is, $|\lambda| = pf + 1$. Because $v_{pf+1} \neq v'_{pf+1}$, we infer that $v'_{pf+1} \neq w$, and thus Lemma 4.27 implies vertex v'_{pf+1} does not correspond to the last element of label λ'; that is, $|\lambda'| > pf + 1$. We conclude that $|\lambda'| > |\lambda|$, and thus

$$b_{\lambda'} = \sum_{j=1}^{|\lambda'|} 2^{\lambda'(j)} = \sum_{j=1}^{|\lambda|} 2^{\lambda'(j)} + \sum_{j=|\lambda|+1}^{|\lambda'|} 2^{\lambda'(j)} > \sum_{j=1}^{|\lambda|} 2^{\lambda'(j)} \geq \sum_{j=1}^{|\lambda|} 2^{\lambda(j)} = b_\lambda$$

(Lemma 7.4 and Claim A.1). □

It follows from Claim A.2 that if $\sigma \neq \sigma'$, then one of the cases handled by Claims A.3 and A.4 must occur. It thus remains to consider the case when $\sigma = \sigma'$.

Claim A.5 If $\sigma = \sigma'$, then either $b_{\lambda'} > b_\lambda$, or $b_{\lambda'} = b_\lambda$ and $crit(\lambda') \geq crit(\lambda)$.

Proof. We prove the claim by first showing that $b_{\lambda'} \geq b_\lambda$ holds, and then showing that $crit(\lambda') \geq crit(\lambda)$ if $b_{\lambda'} = b_\lambda$. Since $\sigma = \sigma'$, it follows that $pf = |\lambda| = |\lambda'|$. Therefore, Claim A.1 implies $\lambda'(j) = \mathrm{VS}(T'_w\langle j-1\rangle) \geq \mathrm{VS}(T_w\langle j-1\rangle) = \lambda(j)$

holds for all j such that $1 \leq j \leq |\lambda|$. Hence, it follows from Lemma 7.4 and the fact $|\lambda'| = |\lambda|$ that

$$b_{\lambda'} = \sum_{j=1}^{|\lambda'|} 2^{\lambda'(j)} \geq \sum_{j=1}^{|\lambda|} 2^{\lambda(j)} = b_\lambda.$$

We conclude the proof by showing that if $b_{\lambda'} = b_\lambda$, then $crit(\lambda') \geq crit(\lambda)$. Suppose that $b_{\lambda'} = b_\lambda$. By definition, the criticalities of labels λ and λ' are equal to the criticalities of vertices $v_{|\lambda|}$ and $v'_{|\lambda'|}$ in trees $T_w \langle |\lambda| - 1 \rangle$ and $T'_w \langle |\lambda'| - 1 \rangle$, respectively. It follows from Lemma 4.27 that $v_{|\lambda|} = v'_{|\lambda'|} = w$. Since $\sigma = \sigma'$, Lemma 2.7 implies tree $T_w \langle |\lambda| - 1 \rangle$ is a subtree of tree $T'_w \langle |\lambda'| - 1 \rangle$. Therefore, all subtrees $R_1, \ldots, R_d$ yielded by vertex w in $T_w \langle |\lambda| - 1 \rangle$ are subtrees of the corresponding subtrees $R'_1, \ldots, R'_d$ yielded by w in $T'_w \langle |\lambda'| - 1 \rangle$; the corresponding subtree is the one with the same root. Lemma 2.1 thus implies $\mathrm{VS}\left(R'_j\right) \geq \mathrm{VS}\left(R_j\right)$ for all j, $1 \leq j \leq d$. Hence, the number of subtrees in $R'_1, \ldots, R'_d$ that have vertex separation at least $\lambda(|\lambda|) = \mathrm{VS}(T_w \langle |\lambda| - 1 \rangle)$ is at least as large as the number of subtrees in $R_1, \ldots, R_d$ that have vertex separation $\lambda(|\lambda|)$. Furthermore, it follows from Lemmas 2.1 and 4.26 that the vertex separation of each tree R'_j is at most $\mathrm{VS}(T'_w \langle |\lambda'| - 1 \rangle) = \lambda'(|\lambda'|)$. We conclude that if $\lambda(|\lambda|) = \lambda'(|\lambda'|)$, the criticality of w in tree $T'_w \langle |\lambda'| - 1 \rangle$ is at least as large as the criticality of w in tree $T_w \langle |\lambda| - 1 \rangle$. But Lemma 7.6 and the fact that $b_{\lambda'} = b_\lambda$ imply $\lambda(|\lambda|) = \lambda'(|\lambda'|)$. Hence, $crit(\lambda') \geq crit(\lambda)$. □

Finally, we combine the results of the previous four claims to prove Lemma 7.7. Consider the sequences σ and σ', integer pf, and tree $T_w \langle j \rangle$ as defined above (page 129). There are two cases to consider, depending on whether $\sigma \neq \sigma'$ or $\sigma = \sigma'$. We first analyze the case when $\sigma \neq \sigma'$. By Claim A.2, σ is not a proper prefix of σ' and σ' is not a proper prefix of σ. Hence, vertex v_{pf+1} exists, since pf is the length of the longest common prefix of σ and σ'. Therefore, the longest common prefix of sequences σ and σ' is a proper prefix of both σ and σ'. If v_{pf+1} is a critical vertex in tree $T_w \langle j \rangle$, then it follows from Claim A.3 that $b_{\lambda'} > b_\lambda$. Otherwise, if v_{pf+1} is noncritical in $T_w \langle j \rangle$, then Claim A.4 implies $b_{\lambda'} > b_\lambda$ as well. Next, we consider the case when $\sigma = \sigma'$. Then by Claim A.5 either $b_{\lambda'} > b_\lambda$ holds, or $b_{\lambda'} = b_\lambda$ and $crit(\lambda') \geq crit(\lambda)$ hold.

Bibliography

[AH73] D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM Journal on Applied Mathematics*, 25(3):403–423, 1973.

[BK96] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996.

[BL91] B. Bollobás and I. Leader. Edge-isoperimetric inequalities in the grid. *Combinatorica*, 11(4):299–314, 1991.

[Bot93] R. A. Botafogo. Cluster analysis for hypertext systems. In *Proceedings of the 16th International ACM SIGIR Conference on Research and Development in Information Retrieval*, Association Methods, pages 116–125, 1993.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, McGraw-Hill, Cambridge, London, 1990.

[CS76] S. Cook and R. Sethi. Storage requirements for deterministic polynomial time recognizable languages. *Journal of Computer and System Sciences*, 13(1):25–37, 1976.

[DPPS01] J. Díaz, M. D. Penrose, J. Petit, and M. Serna. Approximating layout problems on random geometric graphs. *Journal of Algorithms*, 39(1):78–116, 2001.

[DPS02] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

[EST94] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Information and Computation*, 113(1):50–79, 1994.

[Gav97] F. Gavril. Some NP-complete problems on graphs. In *Proceedings of the 11th Conference on Information Sciences and Systems*, pages 91–95, Johns Hopkins University, Baltimore, Maryland, 1997.

[Har64] L. H. Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1):131–135, 1964.

[Kar00] D. R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Journal on Computing*, 29(2):492–514, 2000.

[Kin92] N. G. Kinnersley. The vertex separation number of a graph equals its path-width. *Information Processing Letters*, 42(6):345–350, 1992.

[KP86] L. M. Kirousis and C. H. Papadimitriou. Searching and pebbling. *Theoretical Computer Science*, 47(2):205–218, 1986.

[Lei80] C. E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proceedings of the 21st Symposium on Foundations of Computer Science*, pages 270–281, Syracuse, New York, 1980.

[Len81] T. Lengauer. Black-white pebbles and graph separation. *Acta Informatica*, 16(4):465–475, 1981.

[LT79] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[Möh90] R. H. Möhring. Graph problems related to gate matrix layout and PLA folding. In Gottfried Tinhofer, R. (Rudolf) Albrecht, et al., editors, *Computational Graph Theory*, volume 7 of *Computing Supplementum*, pages 17–51. Springer, 1990.

[MS88] B. Monien and I. H. Sudborough. Min Cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, 58(1-3):209–229, 1988.

[MS89] F. Makedon and I. H. Sudborough. On minimizing width in linear layouts. *Discrete Applied Mathematics*, 23(3):243–265, 1989.

[Mut95] P. Mutzel. A polyhedral approach to planar augmentation and related problems. In Paul G. Spirakis, editor, *Proceedings of the 3rd European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, pages 494–507, Corfu, Greece, 1995. Springer.

[Par76] T. D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. Lick, editors, *Theory and Applications of Graphs*, pages 426–441, Berlin, 1976.

[RS85] N. Robertson and P. D. Seymour. Graph minors—A survey. In *Surveys in Combinatorics*, pages 153–171. Cambridge University Press, 1985.

[Sko00] K. Skodinis. Computing optimal linear layouts of trees in linear time. In M. Paterson, editor, *Proceedings of the 8th European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pages 403–414. Springer-Verlag, 2000.

[TSB01] D. M. Thilikos, M. J. Serna, and H. L. Bodlaender. A polynomial time algorithm for the cutwidth of bounded degree graphs with small treewidth. In F. Meyer auf der Heide, editor, *Proceedings of the 9th European Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 380–390. Springer-Verlag, 2001.

[Yan85] M. Yannakakis. A polynomial algorithm for the min-cut linear arrangement of trees. *Journal of the ACM*, 32(4):950–988, 1985.