Projectivity in Totally Ordered Rooted Trees:

An Alternative Definition of Projectivity and Optimal Algorithms for Detecting Non-Projective Edges and Projectivizing Totally Ordered Rooted Trees

Jiří Havelka

Abstract

This paper discusses the notion of projectivity and algorithms for projectivizing and detecting non-projective edges in totally ordered rooted trees (such trees are used in dependency syntax analysis of natural language, where they are called dependency trees). In the first part, we review the notion of projectivity, then we present a new definition inspired by the algorithmic inquiry and show its equivalence with the classical definitions. We define the canonical projectivization of a totally ordered rooted tree (preserving the tree structure and the relative ordering for all inner nodes and their immediate dependents) and show its uniqueness; we also give a generalization of this result. We then discuss some properties of non-projective edges relevant for the algorithms presented in the following section. In the second part, we present a data representation of totally ordered rooted trees and algorithms for this data representation. The first algorithm computes the projectivization of the input tree, the second algorithm detects non-projective edges of certain types in the input tree (we also give a hint on finding all non-projective edges using its output). Both algorithms can be used for checking projectivity. We prove that the algorithms are optimal: they have time complexities $O(n)$. Furthermore, they can be straightforwardly combined into a single algorithm, preserving the time complexity.

1 Projectivity in Totally Ordered Rooted Trees

This section discusses the condition of projectivity in totally ordered rooted trees. First we give a definition of a totally ordered rooted tree and introduce some notation, then we present the classical definition of projectivity and introduce a new one showing its equivalence, we define the notion of projectivization and show its uniqueness, and finally we divide non-projective edges into three classes and discuss their relationships.

1.1 Totally Ordered Rooted Trees

We give a definition of totally ordered rooted trees, without proofs briefly review some basic properties of rooted trees, and introduce the notation used in this paper.

1.1.1 Definition A *totally ordered rooted tree* is a quadruple $(V,E,r,\leq)$, where (V,E,r) is a rooted tree (V being the finite set of vertices (or nodes), E the set of edges (unordered pairs of nodes), and $r \in V$ the root) and $\leq$ a linear ordering on V. (A totally ordered rooted tree is often called a *dependency tree*.)

In a rooted tree (V,E,r), there is a unique path from the root r to every node a, say $x_0 = r$, $x_1, \ldots,$ $x_n = a$, $n \geq 0$, where $\{x_i, x_{i+1}\} \in E$ for $0 \leq i < n$. Therefore every node a has a uniquely defined *level* equal to the length of the path connecting it with the root, i.e. n, which we will denote $\mathrm{lev}(a)$. For every node $a \neq r$, we will call $b = x_{n-1}$ the *parent* of a (with notation $a \to b$; we will also say that a is a *child* of b or that *a depends on b*). A node with no children is called a *leaf*, a node which is not a leaf is an *internal* node. Obviously, in a rooted tree there is a one-to-one correspondence between the edges and

nodes different from the root (edges correspond uniquely to their "lower" nodes). Nodes with the same parent are called *siblings*. The *height* of a rooted tree is the maximal level occurring in it.

When talking about the tree structure, we will use "vertical-axis" terms such as "above", "below", "upper", "lower" etc., with the root being the highest and the other nodes ordered downwards reversely with respect to their level. (Rooted trees are usually drawn "upside-down" with root at the top and other nodes according to their level downwards, with nodes at the same level in the tree drawn on the same horizontal line.)

The reflexive transitive closure of the relation $\to$ will be denoted $\twoheadrightarrow$; for $a \twoheadrightarrow c$ we will say that c is an *ancestor* of a, or a is a *descendant* of c, or that a is *subordinated* to c. (Note that the relation of dependency $\to$ is irreflexive, whereas the relation of subordination $\twoheadrightarrow$ is defined as reflexive.)

For every node a of a rooted tree $T = (V, E, r)$ we call the tree $T_a = (V_a, E_a, a)$, where $V_a = \{x \in V \mid x \twoheadrightarrow a\}$, $E_a = \{\{x,y\} \in E \mid x, y \in V_a\}$, the *subtree* of T rooted in node a.

When talking about the linear ordering $\leq$ on nodes of a totally ordered rooted tree $(V, E, r, \leq)$, we will use the usual notation $a \geq b$ meaning $b \leq a$, and $a < b$ meaning $a \leq b$ and $a \neq b$ (and similarly for $>$); we will also be using "horizontal-axis" terms such as "left", "right", "in between" etc. with the obvious meaning (we will say that a is to the left from b when $a < b$, etc.).

When drawing totally ordered rooted trees, we accept the following conventions: Nodes are drawn top-down according to their level, with nodes on the same level on the same horizontal line, with the root at the top; nodes are drawn from left to right according to the linear ordering on nodes. Edges are drawn as solid lines.

For an edge $a \to b$ of a totally ordered rooted tree $T = (V, E, r, \leq)$, we call the interval in the linear ordering delimited by the nodes a and b the *span* of the edge $a \to b$.

Please note that the notion of a *totally ordered rooted tree* (cf. Definition 1.1.1) differs from the notion of an *ordered rooted tree*, where for every internal node only a linear ordering of its children is given (i.e. the ordering is not total, it is specified only for sibling nodes). Here we are concerned with rooted trees with a total linear ordering on their nodes.

For the sake of brevity of the definitions in the following section we introduce two predicates:

- A ternary predicate representing the "strictly in between" relation:

  $$\mathrm{Inb}(x,u,v) \stackrel{\mathrm{df}}{=} (u < x \ \&\ x < v) \vee (v < x \ \&\ x < u) \ .$$

  (Obviously, $\mathrm{Inb}(x,u,v)$ should be read as "x lies (strictly) between u and v".)

- A ternary predicate representing the "being siblings" relation:

  $$\mathrm{Sibl}(u,v,b) \stackrel{\mathrm{df}}{=} (u \to b \ \&\ v \to b \ \&\ u \neq v) \ .$$

  ($\mathrm{Sibl}(u,v,b)$ should be read as "u and v are different children of their common parent b".)

We will be taking advantage of the fact that both predicates are symmetric in two of their arguments (Inb in its second and third arguments, Sibl in its first and second arguments).

1.2 Condition of Projectivity for Totally Ordered Rooted Trees

We begin by giving a definition of projectivity using three conditions proved to be equivalent by Marcus (1965) (we take over their denotation), and then present a new condition and prove that it is equivalent to one of the classical ones.

1.2.1 Definition (Marcus (1965)) A totally ordered rooted tree $T = (V, E, r, \leq)$ is *projective* if the following equivalent conditions hold:

(H-H) $\quad (\forall a, b, x \in V)\big(a \to b \ \&\ \mathrm{Inb}(x, a, b) \Longrightarrow x \twoheadrightarrow b\big) \ ,$

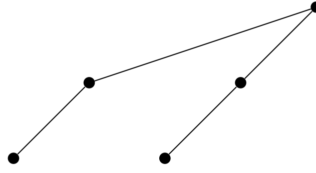

Figure 1: A sample projective tree

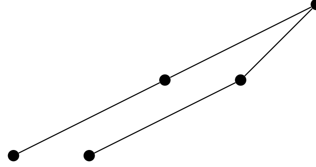

Figure 2: A sample non-projective tree

(L-I) $\quad (\forall a,b,x \in V)\big(a \twoheadrightarrow b \;\&\; \mathrm{Inb}(x,a,b) \Longrightarrow x \twoheadrightarrow b\big)$,

(F) $\quad (\forall u,v,b,x \in V)\big(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \;\&\; \mathrm{Inb}(x,u,v) \Longrightarrow x \twoheadrightarrow b\big)$.

A totally ordered rooted tree not satisfying the conditions is called *non-projective*. (See Figures 1 and 2 for examples of projective and non-projective totally ordered rooted trees, respectively.)

We will not repeat here the proof of the equivalence of the three conditions in Definition 1.2.1, it is quite straightforward and relies on the simple fact that for every two nodes in the relation of subordination there exists a unique finite path between them formed by edges of the rooted tree.

All three conditions in Definition 1.2.1 have in common the following: in a configuration where two (or three) nodes have some structural relationship (i.e. a relationship via the tree structure) and there is a node x between them in the linear ordering, they predicate that the node x be in an analogous structural relationship.

Condition (F) is perhaps most transparent as far as regards the structure of the whole tree. It says that every subtree of a projective tree must be contiguous in the linear ordering. A simple reformulation of the condition (F) gives the following condition of projectivity, which makes this point even more clear.[1]

(F') $\quad (\forall u,v,b \in V)\big(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \Longrightarrow \neg(\exists x \in V)\big(\mathrm{Inb}(x,u,v) \;\&\; x \not\twoheadrightarrow b\big)\big)$.

The condition (F') leads naturally to the notion of a gap in the coverage of a subtree (a gap is the set of "extra-subtree" nodes in the span of the subtree, i.e. between any nodes of the subtree in the linear ordering). Such notion of a gap was used by Holan et al. (1998), who introduce measures of non-projectivity and present a class of dependency-based formal grammars allowing for a varying degree of word-order freedom; Holan et al. (2000) present linguistic considerations concerning Czech and English. In our study, however, we will be concerned with a different notion of a gap.

[1] The equivalence of conditions (F) and (F') is straightforward, see the following first-order-logic reasoning:

$$
\begin{aligned}
&(\forall u,v,b,x \in V)(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \;\&\; \mathrm{Inb}(x,u,v) \Longrightarrow x \twoheadrightarrow b)\\
\Longleftrightarrow\; &(\forall u,v,b,x \in V)(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \Longrightarrow (\mathrm{Inb}(x,u,v) \Longrightarrow x \twoheadrightarrow b))\\
\Longleftrightarrow\; &(\forall u,v,b,x \in V)(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \Longrightarrow (\neg\mathrm{Inb}(x,u,v) \vee x \twoheadrightarrow b))\\
\Longleftrightarrow\; &(\forall u,v,b,x \in V)(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \Longrightarrow \neg(\mathrm{Inb}(x,u,v) \;\&\; x \not\twoheadrightarrow b))\\
\Longleftrightarrow\; &(\forall u,v,b \in V)(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \Longrightarrow (\forall x \in V)(\neg(\mathrm{Inb}(x,u,v) \;\&\; x \not\twoheadrightarrow b)))\\
\Longleftrightarrow\; &(\forall u,v,b \in V)(u \twoheadrightarrow b \;\&\; v \twoheadrightarrow b \Longrightarrow \neg(\exists x \in V)(\mathrm{Inb}(x,u,v) \;\&\; x \not\twoheadrightarrow b)).
\end{aligned}
$$

In a non-projective totally ordered rooted tree, there exists at least one edge $a \to b$ and a node x not satisfying the condition (H-H). We will call such an edge a *non-projective edge* of the totally ordered rooted tree. The set $X_{a \to b} = \{x \in V \mid \mathrm{Inb}(x,a,b)\ \&\ x \not\twoheadrightarrow b\}$ of all nodes causing the non-projectivity of the edge $a \to b$ will be called the *gap* of the edge $a \to b$.

Let us now present in the form of a theorem another condition which is equivalent to the conditions in Definition 1.2.1.

1.2.2 Theorem *A totally ordered rooted tree $T = (V, E, r, \leq)$ is projective if and only if the following condition holds:*

(*) $(\forall a_1, a_2, b, u_1, u_2 \in V)$

$$\Big(\big[a_1 \to b\ \&\ u_1 \twoheadrightarrow a_1\ \&\ \big([a_2 = b\ \&\ u_2 = b] \vee [\mathrm{Sibl}(a_1, a_2, b)\ \&\ u_2 \twoheadrightarrow a_2]\big)\big] \Longrightarrow [a_1 < a_2 \Leftrightarrow u_1 < u_2]\Big)\ .$$

Before giving the proof of this theorem, let us give in words the meaning of condition (*): it says that for any subtree rooted in a node b the relative ordering of all nodes in distinct subtrees of the children of b with respect to each other and to b has to be the same as the relative ordering of b and its children. To put it succinctly, condition (*) requires that the linear ordering of nodes in every subtree respect the ordering of the root and the nodes on the first level of the subtree.

The substantial difference between condition (*) and the conditions in Definition 1.2.1 is that while the conditions in Definition 1.2.1 predicate that nodes in a structural and ordering relationship have a structural relationship, condition (*) predicates that nodes in a structural relationship have an ordering relationship, i.e. the antecedent of condition (*) is concerned only with tree structure, while the consequent only with linear ordering. We will take advantage of this fact when proving theorems and when discussing the algorithms concerning projectivity of totally ordered rooted trees in subsequent sections.

PROOF of Theorem 1.2.2.

(H-H) $\Rightarrow$ (*): We will proceed by contradiction. Let us suppose that the antecedent in the implication in (*) holds, but the consequent does not, and arrive at a contradiction with (H-H).

We will first discuss the case where $a_2 = u_2 = b$. Let us suppose that $a_1 < a_2$ (we can proceed using duality in the opposite case). Then by assumption $u_1 > u_2 = a_2$, and therefore $u_1 \neq a_1$. As $u_1 \twoheadrightarrow a_1$, there exists a path $y_0 = a_1, y_1, \ldots, y_k = u_1$, $k > 0$. Let i be the smallest integer such that $y_i < a_2 < y_{i+1}$. The edge $y_{i+1} \to y_i$ and a_2 are in contradiction with (H-H).

Now let us discuss the remaining case, i.e. suppose that $\mathrm{Sibl}(a_1, a_2, b)\ \&\ u_2 \twoheadrightarrow a_2$. Using the symmetry of Sibl and duality for $<$, let us assume that $a_1 < a_2$ and $u_1 \neq a_1$ (by assumption, for at least one node u_i it has to hold that $u_i \neq a_i$, $i = 1, 2$). There are two cases:

- $u_1 > a_2$: There exists a path $y_0 = a_1, y_1, \ldots, y_k = u_1$, $k > 0$. Let i be the smallest integer such that $y_i < a_2 < y_{i+1}$. The edge $y_{i+1} \to y_i$ and a_2 are in contradiction with (H-H).
- $u_1 < a_2$: By assumption we have $u_2 < u_1$, therefore $u_2 \neq a_2$ and there exists a path $z_0 = a_2, z_1, \ldots,$ $z_l = u_2$, $l > 0$. Let j be the smallest integer such that $z_{j+1} < u_1 < z_j$. The edge $z_{j+1} \to z_j$ and u_1 are in contradiction with (H-H).

(*) $\Rightarrow$ (H-H): Let us again proceed by contradiction. Using duality for $<$, let us suppose that $a \to b$, $a < b$, x is such a node that $a < x < b$, and $x \not\twoheadrightarrow b$. There are two cases to be discussed:

- $b \twoheadrightarrow x$: Then there is a path $x_0 = x, x_1, \ldots, x_n = b$, $n > 0$, with two possibilities: if $x_1 < x$, then the assumption $b > x$ is in contradiction with (*); if $x < x_1$, then the assumption $x > a$ is in contradiction with (*).

- $b \not\twoheadrightarrow x$: Let y be the lowest common ancestor of b and x. Then there are paths $u_0 = y, u_1, \ldots, u_k = b$, $k > 0$, and $v_0 = y$, $v_1, \ldots, v_l = x$, $l > 0$. If $u_1 < v_1$, then the assumption $b > x$ is in contradiction with (*); if $u_1 > v_1$, then the assumption $a < x$ is in contradiction with (*).

This finishes the proof. ∎

After giving the proof of the equivalence of condition (*) and the conditions in Definition 1.2.1, let us remark that the preliminary results concerning this approach to projectivity presented in Veselá, Havelka, and Hajičová (2004), Hajičová et al. (2004), and Veselá and Havelka (2003) contain an error: the condition presented therein and claimed to be equivalent to the conditions discussed above is in fact weaker; the sketches of the algorithm for projectivization, which is to be discussed in detail in the following section, are correct.

The condition presented in the articles has the following form:

$$\text{(P)} \quad (\forall a,b,x \in V)\Big(\big(a \to b \ \&\ a < b \ \&\ x \twoheadrightarrow a \Longrightarrow x < b\big) \ \&\ \big(a \to b \ \&\ a > b \ \&\ x \twoheadrightarrow a \Longrightarrow x > b\big)\Big)$$

We will not prove formally that the condition (P) is weaker than the above discussed conditions of projectivity; this fact follows easily from comparing the condition (P) with the condition (*). For a "counterexample", see Figure 2: it is easy to verify that the totally ordered rooted tree presented there satisfies the condition (P), but is non-projective according to the conditions in Definition 1.2.1 and the condition (*).

1.3 Projectivization of Totally Ordered Rooted Trees

By the canonical projectivization of a totally ordered rooted tree we mean modifying its linear ordering in such a way that for all nodes the relative ordering of the node and its children is preserved. We give the formal definition and show that the total linear ordering of the projectivization is uniquely determined, and furthermore the subtrees of the projectivization are projectivizations of the subtrees. We note that this result straightforwardly generalizes.

1.3.1 Definition For a totally ordered rooted tree $T = (V, E, r, \leq)$, the tree $T^p = (V, E, r, \preccurlyeq)$ is the *canonical projectivization of* T if it is projective and the following condition holds:

$$(\forall a_1, a_2, b \in V)\Big(\big[a_1 \to b \ \&\ (a_2 = b \vee \mathrm{Sibl}(a_1, a_2, b))\big] \Longrightarrow [a_1 < a_2 \Leftrightarrow a_1 \prec a_2]\Big) .$$

For an example of a canonical projectivization, see Figures 1 and 2. The projective totally ordered rooted tree in Figure 1 is the canonical projectivization of the tree in Figure 2.

1.3.2 Theorem *For every totally ordered rooted tree T, there is a unique canonical projectivization T^p of T. Furthermore, for every node a in T, the subtree T^p_a of T^p rooted in a is the canonical projectivization of the subtree T_a of T rooted in a.*

PROOF. We will proceed by induction on the height n of totally ordered rooted trees.

A totally ordered rooted tree of height $n = 0$ consists of a single node, and is therefore its own projectivization; since in such a tree there are no proper subtrees, the projectivization condition for subtrees is vacuously true.

Now let us prove the inductive step: suppose that for every totally ordered rooted tree of height at most n there exists a unique projectivization and the projectivization condition for subtrees holds (i.e. all subtrees of the projectivization are projectivizations of the corresponding subtrees in the original tree). Let $T = (V, E, r, \leq)$ be a totally ordered rooted tree of height $n+1$ and let $a_1, \ldots, a_n$, $n > 0$, be the children of r, $a_1 < \ldots < a_j < r < a_{j+1} < \ldots < a_n$, $1 \leq j \leq n$. From the inductive hypothesis there exist unique projectivizations $T^p_{a_i}$ of subtrees rooted in a_i, $1 \leq i \leq n$, which satisfy the projectivization condition for

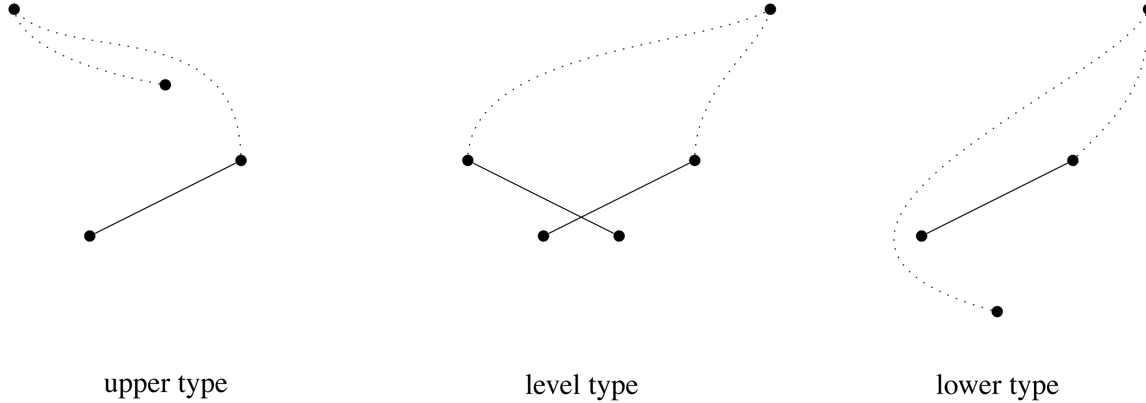

Figure 3: Examples of configurations with non-projective edges

subtrees. Let us consider $T' = (V, E, r, \preccurlyeq)$, where $\preccurlyeq$ is obtained by concatenating the subtrees (i.e. their linear orderings) and the root r in the order $T^p_{a_1}, \ldots, T^p_{a_j}, r, T^p_{a_{j+1}}, \ldots, T^p_{a_n}$. From the definition of $\preccurlyeq$ the tree T' satisfies the condition (*) for the root r, and from the inductive hypothesis the condition also holds for all other nodes in T', hence T' is projective. Obviously, T' also satisfies the projectivization condition for subtrees. Since there is a unique way of satisfying Definition 1.3.1 for $b = r$, the tree T' is uniquely determined, which finishes the proof. ∎

From the proof of Theorem 1.3.2 we see that it can be generalized as follows:

1.3.3 Theorem *Let $T = (V, E, r)$ be a rooted tree. For every node $n \in V$, let there be a linear ordering $\leq_n$ on the set consisting of n and its children. Then there is a uniquely determined projective totally ordered rooted tree $T^p = (V, E, r, \preccurlyeq)$ such that the linear ordering $\preccurlyeq$ respects the linear orderings $\leq_n$ for all nodes $n \in V$ (i.e. for any nodes $a, b \in V$, if their order is determined by some ordering $\leq_n$, $n \in V$, then $a <_n b$ iff $a \prec b$).*

The proof of Theorem 1.3.3 is virtually the same as the proof of Theorem 1.3.2, therefore we do not repeat it here. Obviously, an analogy of the projectivization condition for subtrees holds, too.

1.4 Types of Non-Projective Edges

We discuss certain properties of non-projective edges: we divide them into three classes and show a relationship between the classes which will be used in the section discussing algorithms concerning projectivity and detecting non-projective edges.

1.4.1 Definition Let $T = (V, E, r, \leq)$ be a totally ordered rooted tree, $a \to b$, $a, b \in V$, a non-projective edge, and $X_{a\to b}$ its gap. We will say that the non-projective edge $a \to b$ is of the *lower type*, if $\text{lev}(a) < \min_{x \in X_{a\to b}} \text{lev}(x)$, that it is of the *level type*, if $\text{lev}(a) = \min_{x \in X_{a\to b}} \text{lev}(x)$, and that it is of the *upper type*, if $\text{lev}(a) > \min_{x \in X_{a\to b}} \text{lev}(x)$.

Let us put the definition in words: a non-projective edge is of the lower type if all nodes in its gap are on a strictly lower level than the lower node of the edge; the edge is of the upper type if there is a node in the gap that is on a strictly higher level than the lower node of the edge; and if the highest level achieved by a node in the gap is the same as the level of the lower node of the edge, the edge is of the level type.

Figure 3 schematically shows examples of all above defined types of non-projective edges. The dotted lines represent paths between nodes. All the edges are examples of edges of the corresponding type, but let us stress that they do not represent all possible configurations with non-projective edges. There are many more possible configurations, e.g. for an upper-type non-projective edge the node in the gap of the edge may be an ancestor of the edge's nodes, or a level-type non-projective edge need not cross with another level-type non-projective edge, there may be an upper-type non-projective edge, whose lower node is causing the level-type non-projectivity (the upper node of the level-type non-projective edge in turn causes the upper-type non-projectivity) – see the non-projective tree in Figure 2.

We will now show that the presence of lower-type non-projective edges implies the presence of upper-type non-projective edges. The sample configuration with a lower-type non-projective edge in Figure 3 illustrates well the proof.

1.4.2 Theorem *In a totally ordered rooted tree $T = (V, E, r, \leq)$, for every lower-type non-projective edge $a \rightarrow b$, $a, b \in V$, there is an upper-type non-projective edge $c \rightarrow d$, $c, d \in V$, such that c belongs to the gap $X_{a\rightarrow b}$ and either a or b belongs to the gap $X_{c\rightarrow d}$, and furthermore $\mathrm{lev}(c) = \min_{x \in X_{a\rightarrow b}} \mathrm{lev}(x)$.*

PROOF. Let $c \in X_{a\rightarrow b}$ be an arbitrary maximal node in the gap of the lower-type edge $a \rightarrow b$ (by "maximal" we mean on the highest possible level, i.e. a node such that no strict ancestor of it belongs to the gap $X_{a\rightarrow b}$). Let y be the lowest common ancestor of c and b and let us consider the path $y_0 = y, y_1, \ldots,$ $y_{k-1} = d, y_k = c$, where $k \geq 3$ (because y is on a higher level than b and c on a lower level than a). Since node c is maximal in the gap $X_{a\rightarrow b}$, its parent d is outside the interval delimited by the nodes a and b. Therefore the edge $c \rightarrow d$ is a non-projective edge with either a or b in its gap, and it is of the upper type, since both a and b are on a strictly higher level than c by the assumption that $a \rightarrow b$ is a non-projective edge of the lower type. ∎

From Theorem 1.4.2 we get as a corollary the following important theorem, which allows us to check projectivity of a totally ordered rooted tree by looking only for non-projective edges of certain types and which will be important in our algorithmic inquiries.

1.4.3 Theorem *A totally ordered rooted tree is projective if and only if it contains no non-projective edges of the level and upper types.*

PROOF. Let us consider an arbitrary non-projective totally ordered rooted tree T. According to Theorem 1.4.2, if T contains a non-projective edge of the lower type, then it also contains a non-projective edge of the upper type. Therefore every non-projective totally ordered rooted tree contains at least one non-projective edge of the level or upper type, which is equivalent to the statement of the theorem. ∎

2 Projectivizing Totally Ordered Rooted Trees and Detecting Non-Projective Edges in Totally Ordered Rooted Trees

In this section we present algorithms for canonical projectivization of totally ordered rooted trees and for detecting non-projective edges (and therefore also checking projectivity) in totally ordered rooted trees. First we describe a data representation of totally ordered rooted trees, and then present the algorithms for this data representation and show their complexities.

2.1 Data Representation

The data representation of totally ordered rooted trees described in this section is a minimal representation of both the tree and linear ordering structure of the trees. First we give the data structure for the representation of single nodes and then we describe the requirements we pose on the data representation of the whole totally ordered rooted tree.

sibling	
prev	next
left_child	right_child

Figure 4: Data representation of a node

A node of a totally ordered rooted tree will be represented as an object with fields containing pointers. We will assume that the object contains the following pointer fields:

`left_child`	pointer to the linked list of the node's children to the left from the node, the linked list is ordered inversely to the linear ordering on nodes
`right_child`	pointer to the linked list of the node's children to the right from the node, the linked list is ordered according to the linear ordering on nodes
`sibling`	pointer to a sibling (left sibling if the node is a left child of its parent, right sibling if it is a right child)
`prev`	previous node in the linear ordering
`next`	next node in the linear ordering

We list here only those pointer fields which are indispensable for our discussion of algorithms concerning projectivity in totally ordered rooted trees. In practical application, e.g. a pointer to the parent of each node might be required; furthermore, there would be fields containing any information associated with nodes or edges (represented by their lower nodes).

We adopt the following convention for the pointers: a pointer with no value assigned (i.e. undefined) is considered as a null reference; we do not use a special value for the null reference. The notation `left_child`$[n]$, `prev`$[n]$ etc. is used for the contents of the corresponding fields of the node represented by the pointer n.

The data representation of the whole tree presupposes that the linked list of left (right) children of a node be ordered inversely (according) to the linear ordering on nodes. This is a feature that is usually not present in data representation of totally ordered rooted trees, but transforming different data representations into this one is straightforward. If a different data representation is used, it either has to be transformed into the described representation and the complexity of such a transformation has to be added to the complexity of the algorithms described below, or some of the operations used in the algorithms that are assumed to be atomic have to be implemented in a different way and again the eventual computational cost has to be added to the complexity of the algorithms.

Figure 4 shows pictorially the data representation of a single node. It is used in Figure 5 giving an example of a totally ordered rooted tree and its data representation. Pointers to children are depicted by solid arrows, pointers to siblings by dashed arrows, and pointers representing the linear ordering by dotted arrows.

2.2 Algorithms Concerning Projectivity of Totally Ordered Rooted Trees

In this section we present two algorithms: an algorithm for projectivizing totally ordered rooted trees and an algorithm for detecting non-projective edges of the level and upper types (and therefore checking projectivity) in totally ordered rooted trees. They can be straightforwardly combined into a single algorithm performing both the detection of non-projective edges and the canonical projectivization, which we give in the last subsection.

In the algorithms presented below we take advantage of the fact that totally ordered rooted trees represented using the data structure described above can be traversed in linear time. Specifically, we

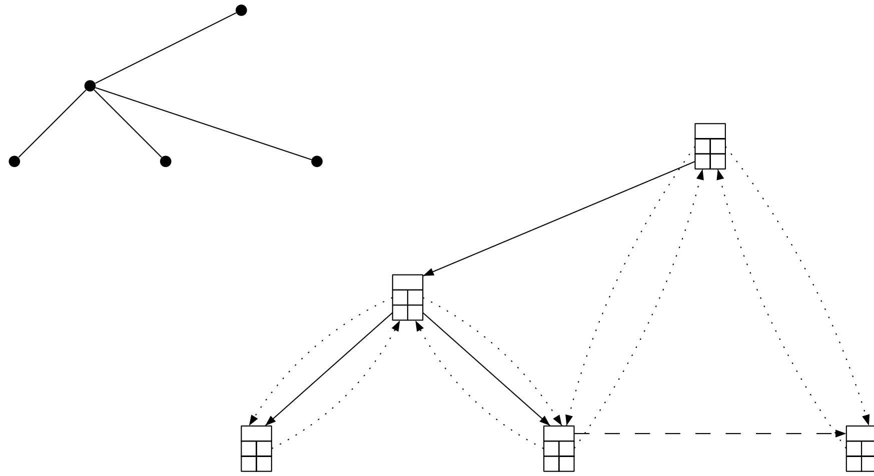

Figure 5: A totally ordered rooted tree and its data representation

will be using traversal by levels from the deepest level to the root, which can be achieved e.g. using a queue as an auxiliary data structure. Here is a sketch how to build such a queue: put the root in the queue and make note that it is on level 0; until you have reached the end of the queue, move to the next node in the queue, append its children (e.g. in the order corresponding to their relative ordering) at the end of the queue and make note that they are one level below their parent. If you want to process nodes by levels bottom up, read the queue backwards.

When talking about complexity bounds, the symbol n will be an integer meaning the size of the input tree (i.e. the number of its nodes). We hope it will not cause any confusion with the use of n as a node variable in the presented algorithms.

2.2.1 Algorithm for Projectivizing

The canonical projectivization of a totally ordered rooted tree was defined in Definition 1.3.1 and in Theorem 1.3.2 we have shown that it always exists and is unique. In Theorem 1.3.3 we have presented a generalization of this result, which applies also to Algorithm 1; it can be easily modified to compute the projectivization given "local" linear orderings, but we will not discuss this modification of the algorithm explicitly.

Algorithm 1 returns for an input totally ordered rooted tree its canonical projectivization. In the process, the original linear ordering on nodes is lost; if we want to preserve it, we have to keep a copy of it.

Let us sketch the idea of the algorithm: the projective ordering of nodes is constructed recursively using only the "structural" pointer fields `left_child`, `right_child` and `sibling` of the data representation; the original ordering contained in the pointers `prev` and `next` is not exploited at all. For each processed node, auxiliary pointers `left_span` and `right_span` are used for the leftmost and rightmost nodes in the subtree of the node, i.e. for its "span". The projective ordering of the processed node's subtree is obtained by concatenating the subtrees of the children of the processed node and the processed node itself in their relative ordering; the auxiliary pointers `left_span` and `right_span` are set accordingly.

Algorithm 1 consists of two subsequent loops over all nodes, the first one on lines 2–21 is the main one, performing the projectivization itself, the second one on lines 22–25 is an auxiliary loop for deleting the auxiliary pointers `left_span` and `right_span`.

In the main loop, the algorithm processes nodes by levels bottom-up – as mentioned above, we are taking advantage of the fact that nodes of a tree can be processed in this way, and lines 1, 2 and 3 represent this traversal decomposed into two embedded loops, the outer one over levels, the inner one over nodes.

Algorithm 1 Projectivize

Input: totally ordered rooted tree T

Output: canonical projectivization of T

```
 1: l_max ← maximal level in T
 2: for l = l_max, ..., 0 do                                   ▷ main loop
 3:     for all nodes n on level l do                          ▷ loop for one level
                                                               ▷ first process left children
 4:         left_span[n] ← n                                   ▷ initialize auxiliary pointer
 5:         d ← n            ▷ auxiliary variable for creating the doubly linked list representing linear ordering
 6:         for all left children c_left of n do               ▷ according to data representation (i.e. in "reverse" order)
 7:             prev[left_span[d]] ← right_span[c_left]
 8:             next[right_span[c_left]] ← left_span[d]
 9:             d ← c_left
10:         end for
11:         left_span[n] ← left_span[d]
                                                               ▷ analogously process right children
12:         right_span[n] ← n                                  ▷ initialize auxiliary pointer
13:         d ← n                                              ▷ auxiliary variable
14:         for all right children c_right of n do             ▷ according to data representation (i.e. in order)
15:             next[right_span[d]] ← left_span[c_right]
16:             prev[left_span[c_right]] ← right_span[d]
17:             d ← c_right
18:         end for
19:         right_span[n] ← right_span[d]
20:     end for
21: end for
22: for all nodes n do                                         ▷ loop for deleting auxiliary pointers
23:     delete left_span[n]
24:     delete right_span[n]
25: end for
```

2.2.1 Theorem *Algorithm 1 returns for any totally ordered rooted tree T the canonical projectivization of T, and its time complexity is $O(n)$.*

PROOF. First let us prove the correctness of the algorithm, i.e. that for any input totally ordered rooted tree it returns its projectivization:

We will prove by induction the following invariant for the inner loop over nodes on individual levels on lines 3–20: after processing each node, the pointers `prev` and `next` of all nodes in the subtree rooted in this node represent the projectivization of the subtree, the linear ordering of the subtree is contiguous, and the pointers `left_span` and `right_span` contain the leftmost and rightmost nodes of the subtree, respectively.

Let us prove the invariant by discussing in detail the inner loop (i.e. lines 4–19) for an individual node. As the loop consists of two sub-blocks, corresponding to left and right children of the processed node, we will go line by line only through the sub-block processing the left children (lines 4–11); the processing of the right children (lines 12–19) is analogous, using duality for linear ordering on nodes.

First, the auxiliary pointer `left_span` of the processed node *n* is initialized to the node itself (line 4) and the auxiliary pointer *d* is initialized to *n* (line 5). The pointer *d* is used in the loop over left children of *n* (lines 6–10) for setting the pointers `prev` and `next` of the doubly linked list representing the linear ordering on nodes.

The doubly linked list representing the linear ordering of the nodes in the subtree rooted in *n* is created using the information already stored at the children of *n* (when processing *n*, all its children have already been processed thanks to traversing the whole tree bottom-up, and so by induction satisfy the invariant). In the loop over left children of *n*, the children are processed according to the data representation (and therefore reversely with respect to the linear ordering on nodes). On lines 7 and 8, the doubly linked lists of the subtrees rooted in q_{eft} and *d* are concatenated (in the first pass, *d* is equal to *n*, thus *n* gets correctly concatenated with the subtree of its first left child thanks to the initialization of `left_span` on line 4). The auxiliary pointer *d* is reset on line 9. After processing all left children of *n*, the auxiliary pointer *d* contains the leftmost child of *n*, and therefore line 11 sets the `left_span` pointer correctly for *n* (if *n* does not have any left children, `left_span` is set correctly thanks to the initialization on line 4).

After the whole inner loop for *n*, the subtree is projective according to condition (*), and since it respects the relative ordering of *n* and all its children, it is the projectivization of the subtree rooted in *n*. Furthermore, the doubly linked list representing the linear ordering is constructed in such a way (subtrees are concatenated) that the linear ordering on the subtree rooted in *n* is contiguous. Thus we have proved the invariant for the algorithm, and so its correctness.

In the loop on lines 22–25, the auxiliary pointers `left_span` and `right_span` are deleted for all nodes.

Now let us show that the time complexity of the algorithm is $O(n)$:

There are two consecutive loops in the algorithm: the main loop on lines 2–21 and the auxiliary loop for deleting the auxiliary pointers on lines 22–25. The time complexity of the auxiliary loop is obviously $O(n)$. Therefore it remains to show the time complexity of the main loop.

The loop on lines 3–20 is processed exactly as many times as there are nodes in the tree. To see that the innermost loops for processing left and right children (lines 6–10 and 14–18, respectively) do not add to the overall complexity, let us observe that the two loops together are processed exactly as many times as there are nodes (except the root). (Every node except the root is a child of some other node, and therefore is processed exactly once by one of these loops). Hence the time complexity of the main loop is $O(n)$.

The overall time complexity of the algorithm is therefore $O(n)$ and the proof is finished. ■

Let us stress that the type of tree traversal used in Algorithm 1 is not essential for the algorithm to work; we use it for the purpose of a smooth incorporation with the algorithm for detecting non-projective edges, described in the next subsection.

As can be readily observed, the only thing needed for the algorithm to work is that children be processed before their parents. This can be most easily done in a post-order traversal, i.e. a depth-first recursive traversal where all children of a node are processed before processing the node itself.

The auxiliary pointers `left_span` and `right_span` can be deleted already in the main loop, but this would only unnecessarily complicate the algorithm due to technicalities.

Algorithm 1 can also serve for linear-time checking of projectivity of the input totally ordered rooted tree. We just have to compare the linear orderings of the original tree and the projectivized one. If they differ, the original tree is non-projective.

The main disadvantage of this way of checking projectivity is that for a non-projective tree, we do not learn much about the causes of its non-projectivity. If we are interested in which edges are non-projective, we have to use some other algorithm.

In the next section we present an efficient algorithm for finding non-projective edges of the level and upper types. We also give a hint on using its output if we want to find all non-projective edges, i.e. also non-projective edges of the lower type.

2.2.2 Algorithm for Detecting Non-Projective Edges

Algorithm 2 detects non-projective edges of the level and upper types (see Section 1.4). From Theorem 1.4.3 we obtain that detecting non-projective edges of these two types is sufficient for checking projectivity. At the end of this section we show how to use the output of Algorithm 2 if we want to find also lower-type non-projective edges.

The algorithm processes the input totally ordered rooted tree by levels bottom up. For all nodes on the processed level, it goes through all edges going down from the nodes (i.e. through all children of the nodes) according to the data representation and checks against the linear ordering on nodes whether there is a node causing non-projectivity of the particular edge of the level or upper type. After processing a level, the nodes one level below it are deleted from the linear ordering on nodes, which allows the check for projectivity to be performed without checking subordination.

For the sake of simplicity of exposition, the algorithm as presented here is destructive in the sense that it destroys the original linear ordering on nodes of the tree. This can be easily remedied either by working with a copy of the whole input tree, or by working with copies of the pointers `prev` and `next` representing the linear ordering for all nodes of the tree.

2.2.2 Theorem *Algorithm 2 returns for any totally ordered rooted tree T the set of all non-projective edges of the level and upper types occurring in T, and its time complexity is $O(n)$.*

PROOF. First let us prove the correctness of the algorithm, i.e. that for any input totally ordered rooted tree it returns all its non-projective edges of the level and upper types:

We will prove the following invariant holding after processing each level of the input tree: the algorithm detects all non-projective edges of the level and upper types whose upper nodes are on this level, all nodes on lower levels are deleted from the linear ordering, and all non-projective edges of the level and upper types whose upper nodes are on higher levels are preserved in the linear ordering (restricted to the nodes on the processed and higher levels).

The algorithm processes nodes by levels bottom-up (as already mentioned above, we are taking advantage of the fact that nodes of a tree can be processed in linear time).

In order to prove the invariant, we have to discuss the main loop on lines 2–30. For nodes on the lowest level the invariant is vacuously true (there are no edges with upper nodes on the lowest level of a tree).

Let us suppose that we process nodes on level l. From the inductive hypothesis we get that in the tree there are only nodes on levels $0,\dots,l+1$ left (in the previous iteration of the main loop nodes on level $l+1$ were processed, and so all nodes on lower levels got deleted) and all non-projective edges of the level and upper types whose upper nodes are on levels $0,\dots,l$ are preserved.

The main loop contains two sub-loops: the loop on lines 3–24 detects the non-projective edges and the loop on lines 25–29 deletes the nodes on the level below the processed one from the linear ordering on nodes.

Now let us discuss in detail the loop for detecting non-projective edges of the level and upper types. In the loop we process all nodes on the current level (the order of processing is irrelevant). For each node n we process all edges going down from this node.

Let us go line by line through the sub-block on lines 4–13 processing the left children of n (i.e. edges going down and left from n); the processing of the right children (lines 14–23) is analogous, using duality for the linear ordering on nodes.

The variable d initialized on line 4 is an auxiliary variable used when detecting non-projectivity of the level and upper types. The variable p initialized on line 5 is a boolean auxiliary variable used for

Algorithm 2 Find non-projective edges of the level and upper types

Input: totally ordered rooted tree T

Output: non-projective edges of the level and upper types in T

```
 1: l_max ← maximal level in T
 2: for l = l_max, ..., 0 do                                   ▷ main loop
 3:     for all nodes n on level l do
                                                    ▷ first process left children
 4:         d ← n             ▷ auxiliary variable for keeping the previously processed node
 5:         p ← true                ▷ suppose that edges to the left are projective
 6:         for all left children c_left of n do   ▷ according to data representation (i.e. in “reverse” order)
 7:             if (c_left = prev[d] & p) then   ▷ edge is projective (because there is no gap), go to the next one
 8:                 d ← c_left                               ▷ keep the processed node
 9:             else   ▷ we found a non-projectivity and mark all remaining edges as non-projective
10:                 p ← false                          ▷ important only in the first pass
11:                 mark c_left as non-projective
12:             end if
13:         end for
                                                    ▷ analogously process left children
14:         d ← n             ▷ auxiliary variable for keeping the previously processed node
15:         p ← true                ▷ suppose that edges to the right are projective
16:         for all right children c_right of n do   ▷ according to data representation (i.e. in order)
17:             if (c_right = next[d] & p) then   ▷ edge is projective (because there is no gap), go to the next one
18:                 d ← c_right
19:             else   ▷ we found a non-projectivity and mark all remaining edges as non-projective
20:                 p ← false
21:                 mark c_right as non-projective
22:             end if
23:         end for
24:     end for
                                   ▷ delete nodes on level l + 1 from the linear ordering
25:     for all nodes n on level l do
26:         for all children c of node n do
27:             delete c from the doubly linked list representing the linear ordering in T
28:         end for
29:     end for
30: end for
```

storing the information whether we have found a non-projective edge (at first we suppose that the edges are projective).

The loop on lines 6–13 processes all left children of n according to the data representation, i.e. reversely with respect to the linear ordering on nodes. The condition on line 7 is crucial: it checks whether the edge $c_{\text{left}} \rightarrow n$ has a gap. If $c_{\text{left}} = \texttt{prev}[d]$ holds (and we have not found a non-projective edge yet, i.e. p is true), we know that all left children up the current child c_{left} form with n a contiguous interval in the linear ordering on nodes. If $c_{\text{left}} \neq \texttt{prev}[d]$, there is some node x between d (either n itself or the previous left child) and the currently processed child c_{left}: from the inductive hypothesis the node x is on level $l+1$ or higher, and obviously it is not subordinated to n, which means that the edge $c_{\text{left}} \rightarrow n$ is a non-projective edge of the level or upper type (its gap containing at least x). When we find a non-projective edge, then all remaining edges going down left from n are also non-projective of the level or upper type (because their gaps contain as a subset the gap of the first detected non-projective edge) – by setting the auxiliary variable p to false, the algorithm marks all remaining edges as non-projective in the subsequent iterations of the loop over left children.

Obviously, in this way we detect all non-projective edges of the level and upper types whose upper nodes are on level l (using the inductive hypothesis that all such non-projectivities are preserved on levels $0, \ldots, l$).

The loop on lines 25–29 deletes the nodes on level $l+1$ from the doubly linked list representing the linear ordering on nodes. Obviously this does not destroy any non-projective edges of the level and upper types with upper nodes on levels $0, \ldots, l-1$, because any such non-projectivity involves nodes on the same or higher levels than the level of the lower node of the non-projective edge. Thus all non-projectivities of the level and upper types with upper nodes on levels $0, \ldots, l-1$ are preserved and all nodes below level l are deleted from the linear ordering on nodes, which finishes the proof of the invariant, and so proves the correctness of the algorithm.

Now let us show that the time complexity of the algorithm is $O(n)$:

The main loop on lines 2–30 contains two sub-loops on lines 3–24 and 25–29. These two sub-loops are processed exactly as many times as there are nodes in the tree. The innermost sub-loops contained in these two loops do not add to the overall complexity, because for both these loops they are processed exactly as many times as there are nodes (except the root).

The overall time complexity of the algorithm is therefore $O(n)$ and the proof is finished. ∎

Let us remark that for Algorithm 2 to work it is essential to process the input totally ordered tree by levels bottom up. To be able to detect all non-projectivities of the level and upper types, we can make do for a non-projective edge of the level or upper types with nodes on the same or higher levels as the lower node of the edge, and removing nodes on the already processed levels from the linear ordering on nodes allows us to avoid performing the subordination check.

No edge from the root can be non-projective (of any type), therefore the main loop of Algorithm 2 over levels on lines 2–30 need not process level 0, i.e. the root. This modification does not however affect the overall time complexity of the algorithm in the general case (although for trees of height 1 it indeed does).

Algorithm 2 does not return the gaps of the detected non-projective edges of the level and upper types. If we want to fully determine the gaps, we can e.g. perform subordination checks for all nodes in the spans of non-projective edges. There are other ways of determining the gaps, but we do not discuss them here. Thanks to the traversal by levels bottom up and taking into account only nodes on the same or higher levels when checking projectivity of an edge, Algorithm 2 need not perform any subordination checks.

From Theorem 1.4.3 we know that for checking projectivity it suffices to look for non-projective edges of the level and upper types. Algorithm 2 detects non-projective edges of the level and upper types, but it does not detect non-projective edges of the lower type.

One way of looking for all non-projective edges can be the naive algorithm, derived directly from the condition (H-H) in Definition 1.2.1, checking for every edge the subordination condition for all nodes in its span. Such algorithm can be straightforwardly implemented with quadratic time complexity (e.g. by pre-computing the subordination relation for all nodes). There are also other possibilities of looking for non-projective edges, but we do not discuss them here.

Another possibility can be to take advantage of the output of Algorithm 2: if we want to find also non-projective edges of the lower type, Theorem 1.4.2 gives us a useful hint. It tells us where to look for lower-type non-projective edges, and moreover which nodes to check for being the cause of a possible lower-type non-projectivity.

Let us suppose that for a totally ordered rooted tree T we know the sets of its upper-type and level-type non-projective edges. Let us consider the set L of all edges $a \to b$ which are neither upper-type nor level-type non-projective and which are on a higher level than some upper-type non-projective edge $c \to d$ such that c is in the span of $a \to b$ and either a or b is in the span of $c \to d$ (obviously we need not include edges from the root); for each edge $a \to b$ in L let $C_{a\to b}$ be the set of all such nodes c. According to Theorem 1.4.2, every lower-type non-projective edge $u \to v$ in T must occur in the set L and for each such edge there is a node in its gap occurring in $C_{u\to v}$.

Algorithm 2 returns all upper-type and level-type non-projective edges, but, however, it does not specify the types of the non-projective edges. We can use the output in conjunction with the naive algorithm as follows:

When constructing the set L, instead of only upper-type non-projective edges, we use all upper-type and lower-type edges $c \to d$. The same applies also to the "candidate" sets $C_{a\to b}$. In this way we get possibly supersets of the sets described above, so the same statements hold for them as well. Now we can use e.g. the naive algorithm for checking projectivity to look for non-projective edges of the lower type just in the set L, and furthermore, for each edge $u \to v$ from L it suffices to perform the subordination check only for nodes from the set $C_{u\to v}$.

The output of Algorithm 2 can be also used in a simpler way: only to construct the set L. We can then proceed by running the naive algorithm on the whole spans of all edges in L.

We have only sketched a way of taking advantage of the output of Algorithm 2 using Theorem 1.4.2, we do not present its detailed analysis.

2.2.3 The Algorithms Combined: Algorithm for Detecting Non-Projective Edges and Projectivizing

Algorithms 1 and 2 presented in previous sections can be straightforwardly combined, preserving the linear time complexity. Since the combination of the algorithms is perspicuous, we will not delve into unnecessary detail and present the resulting algorithm only informally. The properties of the algorithm follow easily from Theorems 2.2.1 and 2.2.2. As noted in Section 2.2.1, the algorithm can be modified to compute the generalized projectivization according to Theorem 1.3.3.

2.2.3 Theorem *Algorithm 3 returns for any totally ordered rooted tree T the canonical projectivization of T (it uses auxiliary pointers* `prev_proj` *and* `next_proj` *for this purpose) and the set of all non-projective edges of the level and upper types occurring in T. Its time complexity is $O(n)$.*

Algorithm 3 performs both the detection of non-projective edges of level and upper types and the projectivization of the input totally ordered rooted tree. As the original ordering gets destroyed during the detection of non-projective edges, the projectivized ordering has to be returned in another doubly linked list, constituted by the pointers `prev_proj` and `next_proj`.

The algorithm consists of two subsequent loops. The loop on lines 2–42 is the main one: the sub-loops on lines 3–20 and 21–25 perform the detection of non-projective edges (they are taken over from Algorithm 2), the sub-loop on lines 26–41 performs the projectivization (it is taken over from Algorithm 1, only the projectivized ordering uses another set of pointers, namely `prev_proj` and

Algorithm 3 Find non-projective edges of the level and upper types and projectivize

Input: totally ordered rooted tree T

Output: non-projective edges of the level and upper types in T and canonical projectivization of T

```
 1: l_max ← maximal level in T
 2: for l = l_max, ..., 0 do                                                    ▷ main loop
 3:     for all nodes n on level l do                  ▷ sub-loop for detecting non-projective edges
 4:         d ← n; p ← true
 5:         for all left children c_left of n do   ▷ according to data representation (i.e. in “reverse” order)
 6:             if (c_left = prev[d] & p) then
 7:                 d ← c_left
 8:             else
 9:                 p ← false; mark c_left as non-projective
10:             end if
11:         end for
12:         d ← n; p ← true
13:         for all right children c_right of n do          ▷ according to data representation (i.e. in order)
14:             if (c_right = next[d] & p) then
15:                 d ← c_right
16:             else
17:                 p ← false; mark c_right as non-projective
18:             end if
19:         end for
20:     end for
21:     for all nodes n on level l do    ▷ sub-loop for deleting nodes on level l+1 from the linear ordering
22:         for all children c of node n do
23:             delete c from the original doubly linked list (pointers prev and next)
24:         end for
25:     end for
26:     for all nodes n on level l do                                      ▷ sub-loop for projectivizing
27:         left_span[n] ← n; d ← n
28:         for all left children c_left of n do   ▷ according to data representation (i.e. in “reverse” order)
29:             prev_proj[left_span[d]] ← right_span[c_left]
30:             next_proj[right_span[c_left]] ← left_span[d]
31:             d ← c_left
32:         end for
33:         left_span[n] ← left_span[d]
34:         right_span[n] ← n; d ← n
35:         for all right children c_right of n do          ▷ according to data representation (i.e. in order)
36:             next_proj[right_span[d]] ← left_span[c_right]
37:             prev_proj[left_span[c_right]] ← right_span[d]
38:             d ← c_right
39:         end for
40:         right_span[n] ← right_span[d]
41:     end for
42: end for
43: for all nodes n do                                         ▷ loop for deleting auxiliary pointers
44:     delete left_span[n]
45:     delete right_span[n]
46: end for
```

next_proj). The second loop of the algorithm on lines 43–46 just deletes the auxiliary pointers used for constructing the projectivization of the input tree (it is taken over from Algorithm 1).

The time complexity $O(n)$ of Algorithm 3 follows easily from the fact that all sub-loops of the main loop on lines 2–42 have overall linear time complexities.

3 Conclusion

We have presented a new approach to projectivity in totally ordered rooted trees. In the first section, we gave a new definition of projectivity and showed its equivalence with the classical ones, defined the notion of canonical projectivization and showed its uniqueness, and defined three types of non-projective edges and discussed their relationship. In the second section, we introduced a data representation of ordered rooted trees and presented two algorithms concerning projectivity: the first one for canonical projectivization, the second one for detecting non-projective edges of the level and upper types, and we also presented their combination performing both tasks. We showed that the time complexities of the algorithms are $O(n)$. We also hinted at a generalization of the algorithm for projectivization and sketched a way of using the output of the algorithm for detecting upper-type and level-type non-projective edges for finding also lower-type non-projective edges.

Our approach originates in the annotation of Topic-Focus Articulation on the tectogrammatical layer of Prague Dependency Treebank 2.0 (in prep.). A short description of the annotator macros in Czech can be found in Veselá and Havelka (2003).

The general framework used in the Prague Dependency Treebank is Functional Generative Description (cf. Sgall, Hajičová, and Panevová (1986) and Hajičová, Partee, and Sgall (1998)). Our results can serve as a mathematical basis for the transition from the surface representation of a sentence to its underlying representation, although the algorithm presented here is to be further adapted to fulfil the linguistic conditions of the transition.

Acknowledgements

The research reported in this article was partially supported by Project No. 1ET201120505 of the Ministry of Education of the Czech Republic.

References

Hajičová, Eva, Jiří Havelka, Petr Sgall, Kateřina Veselá, and Daniel Zeman. 2004. Issues of Projectivity in the Prague Dependency Treebank. *Prague Bulletin of Mathematical Linguistics*, 81:5–22.

Hajičová, Eva, Barbara Partee, and Petr Sgall. 1998. *Topic-Focus Articulation, Tripartite Structures, and Semantic Content*. Kluwer Academic Publishers, Dordrecht, Netherlands.

Holan, Tomáš, Vladislav Kuboň, Karel Oliva, and Martin Plátek. 2000. On Complexity of Word Order. *TAL - Traitement automatique des langues*, 41(1):273–300.

Holan, Tomáš, Vladislav Kuboň, Karel Oliva, and Martin Plátek. 1998. Two Useful Measures of Word Order Complexity. In *Proceedings of the COLING–ACL'98 Workshop on Dependency-Based Grammars*, Montréal. Université de Montréal.

Marcus, Solomon. 1965. Sur la notion de projectivité [on the notion of projectivity]. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 11:181–192.

Prague Dependency Treebank 2.0. in prep. http://ufal.mff.cuni.cz/pdt2.0/.

Sgall, Petr, Eva Hajičová, and Jarmila Panevová. 1986. *The Meaning of the Sentence and Its Semantic and Pragmatic Aspects*. Academia/Reidel Publishing Company, Prague, Czech Republic/Dordrecht, Netherlands.

Veselá, Kateřina, Jiří Havelka, and Eva Hajičová. 2004. Condition of Projectivity in the Underlying Dependency Structures. In *Proceedings of the 20th International Conference on Computational Linguis-*

tics, volume I, pages 289–295, Geneva, Switzerland, August 23-27. Association for Computational Linguistics. ISBN 1-932432-48-5.

Veselá, Kateřina and Jiří Havelka. 2003. Anotování aktuálního členění věty v Pražském závislostním korpusu [Annotation of Topic-Focus Articulation in the Prague Dependency Treebank]. Technical report, ÚFAL/CKL MFF UK, December 2003.