

# Phrase Projectivity in Antigone

Jonathan Sterling

April 2013

## 1 Dependency Trees and Their Projectivity

A dependency tree encodes the head-dependent relation for a string of words, where arcs are drawn from heads to their dependents. We consider a phrase *projective* when these arcs do not cross each other, and *discontinuous* to the extent that any of the arcs intersect. Figure 1 illustrates the various kinds of projectivity violations that may occur.

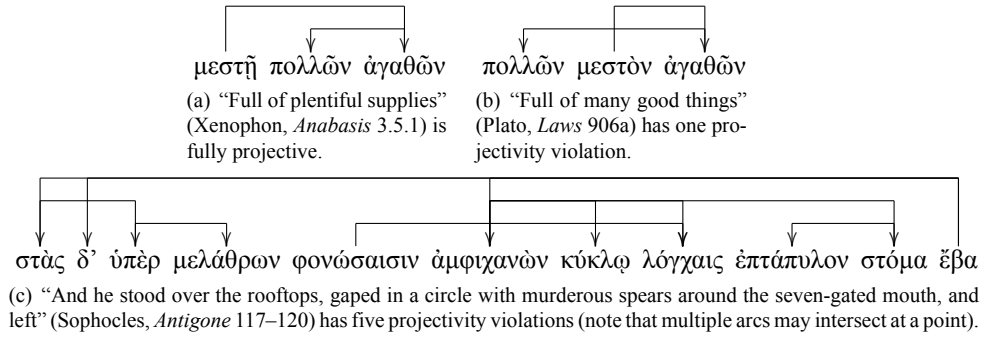


Figure 1: A dependency path wrapping around itself is a projectivity violation, as in (b); interlacing adjacent phrases also violate projectivity, as in (c). Examples (a–b) drawn from Devine & Stephens.

In this paper, we use a concrete metric of projectivity  $\omega$ , given by the following ratio:

$$\omega = \frac{\text{number of violations}}{\text{number of arcs}}$$

Section 2 deals with the development of an algorithm to compute this quantity for a particular dependency tree.

## 2 The Algorithm

Dependency trees are a recursive data structure with a head node, which may have any number of arcs drawn to further trees (this is called a *rose tree*). We represent them as a Haskell data-type as follows:

```
data Tree  $\alpha$  =  $\alpha \curvearrowright [\text{Tree } \alpha]$ 
```

This can be read as “For all types  $\alpha$ , a Tree of  $\alpha$  is constructed from a label of type  $\alpha$  and a forest of Trees of  $\alpha$ ,” where brackets are a notation for lists.

Given a tree, we can extract its root label by pattern matching on its structure as follows:

```
getLabel :: Tree  $\alpha$   $\rightarrow$   $\alpha$ 
getLabel ( $l \curvearrowright \_$ ) =  $l$ 
getForest :: Tree  $\alpha$   $\rightarrow$  [Tree  $\alpha$ ]
getForest ( $\_ \curvearrowright ts$ ) =  $ts$ 
```

## 2.1 From Edges to Trees

A sentence from the Perseus treebank is in the form of a list of words that are indexed their linear position, and cross-referenced by the linear position of their dominating head. We shall consider each index to be a *vertex*, and each pair of vertices to be an Edge, which we shall write as follows:

```
data Edge  $\alpha$  =  $\alpha \leftrightarrow \alpha$  deriving Eq
```

An Edge  $\alpha$  is given by two vertices of type  $\alpha$ ; the **deriving** Eq statement generates the code that is necessary to determine whether or not two Edges are equal using the ( $\equiv$ ) operator. In order to perform our analysis, we should wish to transform the raw list of edges into a tree structure. The basic procedure is as follows:

First, we try to find the root vertex of the tree. This will be a vertex that is given as the head of one of the words, but does not itself appear in the sentence:

```
rootVertex :: Eq  $\alpha \Rightarrow$  [Edge  $\alpha$ ]  $\rightarrow$  Maybe  $\alpha$ 
rootVertex  $es$  = find ( $\notin$   $deps$ )  $heads$  where
   $heads$  = ( $\lambda(x \leftrightarrow y) \rightarrow x$ )  $\langle \$ \rangle$   $es$ 
   $deps$  = ( $\lambda(x \leftrightarrow y) \rightarrow y$ )  $\langle \$ \rangle$   $es$ 
```

If the data that we are working with are not well-formed, there is a chance that we will not find a root vertex; that is why the type is given as Maybe.

Then, given a root vertex, we look to find all the edges that it touches, and try to build the subtrees that are connected with those edges.

```
onEdge :: Ord  $\alpha \Rightarrow$   $\alpha \rightarrow$  Edge  $\alpha \rightarrow$  Bool
onEdge  $i$  ( $x \leftrightarrow y$ ) =  $x \equiv i \vee y \equiv i$ 
oppositeVertex :: Ord  $\alpha \Rightarrow$   $\alpha \rightarrow$  Edge  $\alpha \rightarrow$   $\alpha$ 
oppositeVertex  $i$  ( $x \leftrightarrow y$ )
  |  $x \equiv i$       =  $y$ 
  | otherwise =  $x$ 
```

This is done recursively until the list of edges is exhausted and we have a complete tree structure:

```

treeFromEdges :: Ord  $\alpha$   $\Rightarrow$  [Edge  $\alpha$ ]  $\rightarrow$  Maybe (Tree  $\alpha$ )
treeFromEdges es = buildWithRoot es ($) rootVertex es where
  buildWithRoot es root = root  $\curvearrowright$  sortedChildren where
    roots          = oppositeVertex root ($) filter (onEdge root) es
    children       = buildWithRoot (filter ( $\neg$   $\circ$  onEdge root) es) ($) roots
    sortedChildren = sortBy (compare 'on' getLabel) children

```

## 2.2 Counting Violations: Computing $\omega$

Violations are given as an integer tally:

```
type Violations = Integer
```

The basic procedure for counting projectivity violations is as follows: flatten down the tree into a list of edges cross-referenced by their vertical position in the tree; then traverse the list and see how many times these edges intersect each other.

```
type Level = Integer
```

The vertical position of a node in a tree is represented as its Level, counting backwards from the total depth of the tree. That is, the deepest node in the tree is at level 0, and the highest node in the tree is at level  $n$ , where  $n$  is the tree's depth.

```

levels :: Tree  $\alpha$   $\rightarrow$  [[ $\alpha$ ]]
levels t = map (map getLabel) $
  takeWhile ( $\neg$   $\circ$  null) $
    iterate ( $\gg$  getForest) [t]

```

```

depth :: Tree  $\alpha$   $\rightarrow$  Integer
depth = genericLength  $\circ$  levels

```

We can now annotate each node in a tree with what level it is at:

```

annotateLevels :: Tree  $\alpha$   $\rightarrow$  Tree (Level,  $\alpha$ )
annotateLevels tree = aux (depth tree) tree where
  aux l (x  $\curvearrowright$  ts) = l, x  $\curvearrowright$  aux (l - 1) ($) ts

```

Then, we fold up the tree into a list of edges and levels:

```

allEdges :: Ord  $\alpha$   $\Rightarrow$  Tree  $\alpha$   $\rightarrow$  [(Level, Edge  $\alpha$ )]
allEdges tree = aux (annotateLevels tree) where
  aux (_, x  $\curvearrowright$  ts) = ts  $\gg$  go where
    go t@(l, y  $\curvearrowright$  _) = (l, edgeWithRange [x, y]) : aux t

```

```

edgeWithRange :: (Foldable  $\phi$ , Ord  $\alpha$ )  $\Rightarrow$   $\phi$   $\alpha$   $\rightarrow$  Edge  $\alpha$ 
edgeWithRange xs = minimum xs  $\leftrightarrow$  maximum xs

```

A handy way to think of edges annotated by levels is as a representation of the arc itself, where the ends of the edge are the endpoints, and the level is the height of the arc.

If one end of one arc is between the ends of another, then there is a single intersection. If one arc is higher than another and the latter is in between the endpoints of the former, there is no violation; but if they are at the same level, or if the latter is higher than the former, there is a double intersection. Otherwise, there is no intersection.

```

checkEdges :: Ord α ⇒ (Level, Edge α) → (Level, Edge α) → Violations
checkEdges (l, xy@(x ↔ y)) (l', uv@(u ↔ v))
  | y ∈R uv ∧ u ∈R xy          = 1
  | x ∈R uv ∧ v ∈R xy          = 1
  | x ∈R uv ∧ y ∈R uv ∧ l ≥ l' = 2
  | u ∈R xy ∧ v ∈R xy ∧ l ≤ l' = 2
  | otherwise                    = 0

```

We determine whether a vertex is in the bounds of an edge using  $\cdot \in_R \cdot$ .

```

· ∈R · :: Ord α ⇒ α → Edge α → Bool
z ∈R x ↔ y = z > minimum [x, y] ∧ z < maximum [x, y]

```

We can now use what we've built to count the intersections that occur in a collection of edges. This is done by adding up the result of checkEdges of combination of each edge with the subset of edges which are at or below its level:

```

edgeViolations :: Ord α ⇒ [(Level, Edge α)] → Violations
edgeViolations xs = sum $ violationsWith xs where
  rangesBelow (l, _) = filter (λ(l', _) → l' ≤ l) xs
  violationsWith x    = sum $ (checkEdges x <$> rangesBelow x)

```

Finally,  $\omega$  is computed for a tree as follows:

```

ω :: (Fractional n, Ord α) ⇒ Tree α → n
ω tree = violations / totalArcs where
  edges      = allEdges tree
  totalArcs  = genericLength edges
  violations = fromInteger $ edgeViolations edges

```