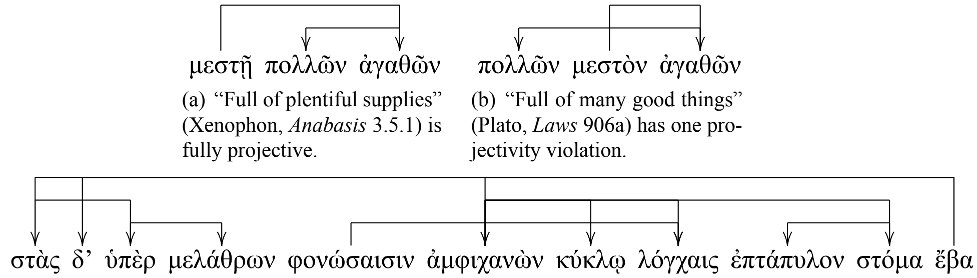
A Survey of Phrase Projectivity in Antigone

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.


μεστῇ πολλῶν ἀγαθῶν
πολλῶν μεστὸν ἀγαθῶν
στὰς δ’ ὑπὲρ μελάθρων φονώσαισιν ἀμφιχανὼν κύκλῳ λόγχαις ἑπτάπυλον στόμα ἔβα

(a) “Full of plentiful supplies” (Xenophon, *Anabasis* 3.5.1) is fully projective.

(b) “Full of many good things” (Plato, *Laws* 906a) has one projectivity violation.

(c) “And he stood over the rooftops, gaped in a circle with murderous spears around the seven-gated mouth, and left” (Sophocles, *Antigone* 117–120) has five projectivity violations (note that multiple arcs may intersect at a point).

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 ω, 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 Algorithm & Data Representation

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 α = α ↷ [Tree α]
```

This can be read as "For all types α, a Tree of α is constructed from a *label* of type α and a *subforest* of Trees of α," where brackets are a notation for lists.

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

```
getLabel :: Tree α → α
getLabel (l ↷ _) = l
getForest :: Tree α → [Tree α]
getForest (_ ↷ 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 by 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 α = α ↔ α deriving Eq
```

An Edge α is given by two vertices of type α; 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 α ⇒ [Edge α] → Maybe α
rootVertex es = find (∉ deps) heads where
  heads = ⟦ (λ(x ↔ y) → x) es ⟧
  deps  = ⟦ (λ(x ↔ y) → y) 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 :: Eq α ⇒ α → Edge α → 𝔹
onEdge i (x ↔ y) = x ≡ i ∨ y ≡ i

oppositeVertex :: Eq α ⇒ α → Edge α → α
oppositeVertex i (x ↔ y)
   | x ≡ i       = y
   | otherwise = x
```

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

```
treeFromEdges :: Ord α ⇒ [Edge α] → Maybe (Tree α)
treeFromEdges es = ⟦ (buildWithRoot es) (rootVertex es) ⟧ where
  buildWithRoot es root = root ↷ sortedChildren where
    roots            = ⟦ (oppositeVertex root) localVertices ⟧
    children         = ⟦ (buildWithRoot foreignVertices) roots ⟧
    localVertices    = filter (onEdge root) es
    foreignVertices  = filter (¬ ∘ onEdge root) es
    sortedChildren   = sortBy (compare ‘on‘ getLabel) children
```

2.2 Counting Violations: Computing ω

Violations are given as an integer tally:

```
type Violations = ℤ
```

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 = ℤ
```

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 α → [[α]]
levels t = fmap (fmap getLabel) $
  takeWhile (¬ ∘ null) $
  iterate (≫=getForest) [t]


depth :: Tree α → ℤ
depth = length ∘ levels
```

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

```
annotateLevels :: Tree α → Tree (Level, α)
annotateLevels tree = aux (depth tree) tree where
  aux l (x ↷ ts) = (l, x) ↷ ⟦ (aux (l − 1)) ts ⟧
```

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

```
allEdges :: Ord α ⇒ Tree α → [(Level, Edge α)]
allEdges tree = aux (annotateLevels tree) where
  aux ((_, x) ↷ ts) = ts ≫= go where
    go t@((l, y) ↷ _) = (l, edgeWithRange [x, y]) : aux t
```

```
$\mathsf{edgeWithRange} :: (\mathsf{Ord}\ \alpha) \Rightarrow [\alpha] \to \mathsf{Edge}\ \ \alpha$
$\mathsf{edgeWithRange}\ xs = \mathsf{minimum}\ xs \leftrightarrow \mathsf{maximum}\ xs$
```

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

If one end of an 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.

```
$\mathsf{checkEdges} :: \mathsf{Ord}\ \alpha \Rightarrow (\mathsf{Level}, \mathsf{Edge}\ \alpha) \to (\mathsf{Level}, \mathsf{Edge}\ \alpha) \to \mathsf{Violations}$
$\mathsf{checkEdges}\ (l, xy@(x \leftrightarrow y))\ (l', uv@(u \leftrightarrow v))$
    $|\ y \in_E uv \wedge u \in_E xy$             $= 1$
    $|\ x \in_E uv \wedge v \in_E xy$             $= 1$
    $|\ x \in_E uv \wedge y \in_E uv \wedge l \geqslant l'$ $= 2$
    $|\ u \in_E xy \wedge v \in_E xy \wedge l \leqslant l'$ $= 2$
    $|\ otherwise$                     $= 0$
```

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

```
$\cdot \in_E \cdot :: \mathsf{Ord}\ \alpha \Rightarrow \alpha \to \mathsf{Edge}\ \alpha \to \mathbb{B}$
$z \in_E x \leftrightarrow y = z > \mathsf{minimum}\ \ [x, y]$
                 $\wedge\ z < \mathsf{maximum}\ [x, y]$
```

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 the combination of each edge with the subset of edges which are at or below its level:

```
$\mathsf{edgeViolations} :: \mathsf{Ord}\ \alpha \Rightarrow [(\mathsf{Level}, \mathsf{Edge}\ \alpha)] \to \mathsf{Violations}$
$\mathsf{edgeViolations}\ xs = \mathsf{sum}\ [\![\ \mathsf{violationsWith}\ xs\ ]\!]$ where
   $\mathsf{rangesBelow}\ (l, \_) = \mathsf{filter}\ (\lambda(l', \_) \to l' \leqslant l)\ xs$
   $\mathsf{violationsWith}\ x\ \ \ = \mathsf{sum}\ [\![\ (\mathsf{checkEdges}\ x)\ (\mathsf{rangesBelow}\ x)\ ]\!]$
```

Finally, ω is computed for a tree as follows:

```
$\omega :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Tree}\ \alpha \to \mathbb{Q}$
$\omega\ tree = \dfrac{\mathsf{violationsCount}}{\mathsf{totalArcsCount}}$ where
   $\mathsf{edges}$           $= \mathsf{allEdges}\ tree$
   $\mathsf{violationsCount} = \mathsf{fromIntegral}\ (\mathsf{edgeViolations\ edges})$
   $\mathsf{totalArcsCount}$  $= \mathsf{length\ edges}$
```

3 Parsing the Perseus Treebank

The Persues treebank is a collection of XML files, which have data in the following (simplified) scheme:

```
<sentence id="2900759">
  <word id="1" form="χρὴ" lemma="χρή" head="0" />
  <word id="2" form="δὲ" lemma="δέ" head="1" />
  ...
</sentence>

<sentence id="2900760">
  <word id="1" form="μεγάλοι" lemma="μέγας" head="3" />
  <word id="2" form="δὲ" lemma="δέ" head="12" />
  ...
</sentence>
```

We can express the general shape of such a document as follows:

```
newtype XML      = XML [Content]
newtype Word     = Word Element
newtype Sentence = Sentence Element
```

To convert XML into trees, we must first extract the sentences from the file, and then we convert those into trees.

```
sentencesFromXML :: XML → [Sentence]
sentencesFromXML (XML xml) = do
  elems ← onlyElems xml
  ⟦ Sentence (findElements (simpleName "sentence") elems) ⟧
```

To build a tree from a sentence, first we get all of the words from that sentence and convert them into edges.

```
wordsFromSentence :: Sentence → [Word]
wordsFromSentence (Sentence s) = ⟦ Word (findChildren (simpleName "word") s) ⟧
```

Edges are the content of the `head` attribute paired with that of the `id` attribute.

```
edgeFromWord :: Word → Maybe (Edge ℤ)
edgeFromWord (Word w) = ⟦ (readAttr "head" w) ↔ (readAttr "id" w) ⟧
```

Thence, we can build a tree from a sentence.

```
treeFromSentence :: Sentence → Maybe (Tree ℤ)
treeFromSentence = treeFromEdges ∘ edgesFromSentence where
  edgesFromSentence :: Sentence → [Edge ℤ]
  edgesFromSentence s = catMaybes ⟦ edgeFromWord (wordsFromSentence s) ⟧ where
```

By putting the pieces together, we also derive a function to read all the trees from an XML document:

```
treesFromXML :: XML → [Tree ℤ]
treesFromXML xml = catMaybes ⟦ treeFromSentence (sentencesFromXML xml) ⟧
```

Finally, we must read the file as a string, parse it as XML, and then convert that XML into a series of trees.

```
treesFromFile :: FilePath → IO [Tree ℤ]
treesFromFile path = ⟦ (treesFromXML ∘ XML ∘ parseXML) (readFile path) ⟧
```

4 Analysis of Data

We compute the average ω of the trees contained in a file as follows:

```
analyzeFile :: FilePath → IO ℚ
analyzeFile path = do
  trees ← treesFromFile path
  return (average ⟦ ω trees ⟧)
```

Appendix: Auxiliary Functions

```
simpleName :: String → QName
simpleName s = QName s Nothing Nothing
```

```
readAttr :: Read α ⇒ String → Element → Maybe α
readAttr n = fmap read ∘ findAttr (simpleName n)
```

```
average :: Fractional n ⇒ [n] → n
                sum xs
average xs = ─────────
              length xs
```