conj_head

Observations Related to the Problem Statement

The problem of identifying the coordinating conjunction, and separating it from subordinating conjunction is a problem in itself, with the boundaries between the two sometimes not being explicit. Nonetheless, in our treatment of the problem, we shall identify coordinating conjunctions with their POS tag (`CCONJ`), and limiting ourselves to a particular deprel, `cc`. Notice that this distinction is necessary, and needs to be marked explicitly owing to the problem related to multiple deprels being associated to the POS tag. We take a look at the different issues associated with the problem that makes it a difficult one to solve in following subsections.

Direction of Dependency

One of the intuitive methods of approaching the problem at hand is to isolate the problematic token in a tree, and then check if the dependency edge to this token is in the correct direction. However, the identification of the correct direction can be non-trivial if worked in a language-independent manner. Consider the case of `sa`, and how it differs from `en`, as in Example 1. In `en`, the coordinating conjunction occurs in between the different conjuncts. In the given example for `sa`, the conjunction is linked with the last conjunct in a form that is typical of the language.

Example 1.
Text (`sa`): तस्य त्रयः पुत्राः परमदुर्मेधसः वसुशक्तिः उग्रशक्तिः अनन्तशक्तिश्च इति बभूवुः ।
Translit: *tasya trayaḥ putrāḥ paramadurmedhasaḥ vasuśaktiḥ ugraśaktiḥ* ***anantaśaktiśca*** *iti babhūvuḥ .*
Lit.: His three sons extremely-stupid Vasushakti Ugrashakti **Anantashakti-and** known-by-these-names there-were .
Translated: There were his three extremely stupid sons, called Vasushakti, Ugrashakti, and Anantashakti.

Note how in the data, the coordinating conjunction च (*ca*; and) is attached to the last conjunct, unlike in English where the conjunction (*and*) exists as a token on its own. It is also worth pointing out that the token referred to above is not the only coordinating conjunction in the language, with other conjunctions may/may not be attached to the last conjunct.

We have not yet talked about the case of RTL languages. Consider the case of Hebrew, for example. Written in right-to-left manner, the token for coordinating conjunction is added on to the next word as a prefix. However, this conjunction token is not the only prefix used in the language, and a singular word can have multiple prefixes. It would still have been possible to isolate the prefix if the associated character were reserved only for prefixes of such nature. However, the character in question can also occur as the first character in a word, without implying conjunction. The same process is elaborated in Example 2. The associated character is ו (*w*).

Example 2.
Text (he): התפוח והכלב; ורד; הייתי בטיול ובגדול נהנתי
Translit: *ht̲pūwḥ* ***w****hk̲lb̲; **w**rd̲d; ht̲ b̲ṭwl **wb̲ḡd̲wl** nhnt̲*;
Lit.: the-apple and-the-dog; Rose; I-was on-a-trip **and-in-large** I-enjoyed;
Translated: The apple and the dog; Rose; I was on a trip and in general I liked it;

Although the problem may seem complex, it is not so. Effectively, we can consider the cases of `sa` and `he` as similar, differing only in aspect of prefix, or as suffix. Once we are able to identify the relevant affix, the problem can be simplified to that of direction problem. Notice that the identification of the relevant affix is a tokenisation problem, which is outside the scope of this research. In our treatment of the languages, we assume (and are given in CONLL-U representation) the syntactic tokens split into the smaller syntactic words.

Asyndetic Coordination

Asyndetic coordination refers to the case where the coordinating conjunction is omitted. A typical example of this is listed below, where comma (or some other punctuation) delimits the different tokens, and acts as conjunction marker. While this may be frequent in some languages, the alternative approach of using a conjunction between every conjunct is also possible.

Example 3.
Asyndetic: A, B, and C.
Notice the lack of a conjunction between A and B.
Non-Asyndetic: A and B and C.
The conjunction (and) is present in between every conjunct.

In either of the case, we need to restrict our focus on the present coordinating conjunction, and make sure the conjunction is linked to the next conjunct. The problem here remains the same, making sure the conjunction is attached in the right direction.

Nested Conjunctions

It can be argued that nested conjunctions can not be handled the same way as the other conjunctions in the scope of the problem. We use the examples as given in UD guidelines on the problem[1], without adding conjunctions in between the tokens.

Example 4.

1. A, B, C
2. (A, B), C
3. A, (B, C)

[1] `https://universaldependencies.org/u/dep/all.html#nested-coordination`

Without using the enhanced dependencies, the trees for the first two examples cannot be distinguished in the trees. Only the last example can be distinguished from the first two. However, if there are conjunctions present, all the three cases are distinguishable from each other.

It is important to note that we work with the hypothesis that the conjunction is located always close to the conjuncts. This is intuitive, but in the event of this being not the case, the conjunction will introduce a non-projectivity into the sentence. We do not want to introduce non-projectivities in the sentence where it was not already, although it might happen that we end up removing some of the non-projectivities in the process.

Conjunction Sandwich

We have so far discussed only the cases where the direction of dependency is wrong. Such cases are easily detectable. However, there is one more case which is significantly more difficult to determine. Consider the example of a subtree in Figure 1. Notice that token 'B' is the conjunction, while 'A', and 'C' are conjuncts. The conjunction 'B' should be correctly linked to 'C'. The node 'D' refers to the shared parent of nodes 'A', 'B', 'C', or the root of the sentence, as the case might be.

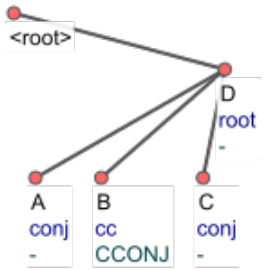


Figure 1: Possible Case of Conjunction Sandwich

In the above figure, we can observe that the direction of the association of the conjunction is correct. However, that does not mean that it is linked to the correct head. This problem can be present in the default annotation, or might be introduced after the tree has been corrected for the misdirected dependency. We refer to such cases as a Conjunction Sandwich, since the conjunction is sandwich-ed in between the conjuncts, with no way of knowing the conjuncts. In the figure, we explicitly mention that the surrounding tokens (or subtree heads) are conjuncts. However, it is fair to assume that neither of the two would be labelled by the deprel, and that makes such cases even harder to detect. In our experiments, we tried to detect such cases, without any success. We therefore do not deal with such cases in this research.

Dataset Definition

The experiment was initially started on UDv2.3, but owing to the release of UDv2.4 in May 2019, the experiment was transported entirely to UDv2.4. It is worth noting that there were far more cases of this problem being identified in

UDv2.3, rather than in UDv2.4. Nonetheless, there exist significant cases of the problem in UDv2.4 as well.

We limit our treatment of the problem to `af`, and `ar`. The treebank for working with `sa` is too small, and so we discard it from the dataset, owing to the smaller size of the treebank. The other languages are not considered since the number of erroneous instances with respect to attachment in wrong direction is too small. Note that we don't conduct any experiments on agglutinative languages, owing to the complexity of the resulting agglutinated token. Again, as in case of `sa` and `he`, it should be possible however to do so once the agglutinates have been identified, and isolated.

With respect to treebanks, `ar` has 2 different treebanks, not including `ar`-PUD. We focus our attention on `ar`-PADT treebank here because the other treebank for the language is delexicalised, and requires a license in order to obtain the textual data.

Experimental Setup

We start by identifying the cases where the direction of dependency is inverted. Once such cases have been identified, we start by flipping the direction of attachment, towards the most likely conjunct, as defined in previous section.

While `ar` is a RTL language, it is written in CONLL-U format in LTR format. As such, given that `af` is already an LTR language, the algorithm for identification, and correction of the direction of attached dependency is the same for both languages. Table 1 lists the counts of instances identified as attached in the wrong direction, in comparison to the total number of instances.

Language	Misdirected	Total	Percentage
`af`	1 829	1 832	99.836
`ar`	1 411	13 855	10.184

Table 1: Counts of Coordinating Conjunctions attached in wrong direction

We limit our treatment of the problem to the case where we do not need to change the level in the tree (where the node is incorrectly attached) by more than 1. In essence, the wrong attachment of the conjunction can be corrected by finding a conjunct that is in the same level as the wrong conjunct, or is within the subtree of this wrong conjunct. In very rare cases, it might be necessary to attach the conjunction to the parent of the wrong conjunct it is attached to. If none of these is valid, we hypothesize that the annotation for the sentence was faulty, and thus it requires manual inspection for it to be corrected.

With respect to the above statement about the change of level being restricted to a maximal value of 1, there are 3 trees possible, as shown in Figures 2, and 3. The tokens follow the same conventions as discussed in previous subsection on Conjunction Sandwich.

Notice that while the 3 cases are separate, there is no deterministic way of knowing what case an instance might refer to. As such, we handle all the 3 cases in decreasing order of priority, i.e. we try to handle the case as in Figure 2 first, failing which we try to solve it with respect to the case as in Figure 3a, and

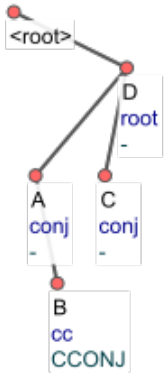
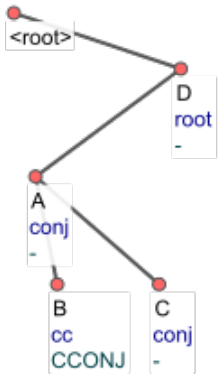
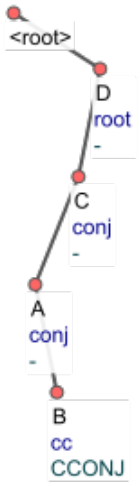


Figure 2: Possible Wrong Attachments of a Coordinating Conjunction: Both conjuncts at same level



(a) Wrong conjunct at higher level



(b) Correct conjunct at higher level

Figure 3: Possible Wrong Attachments of a Coordinating Conjunction

eventually as in Figure 3b. If a particular instance is still not corrected after the consideration of the last case, we leave it untouched. We describe in detail the algorithm for the task in the next subsection.

Algorithm

We start with defining some wrapper functions in Algorithm 1 and 2. While the first one checks for the coordinating conjunctions that are attached in wrong direction, the second one tries to change the parent of the given node x to a new parent z. In case the new association would be non-projective, the function rolls back to the previous parent. If the projectivity is preserved, the function returns a **true** value, which allows us to terminate the function whenever the function call is made inside another function. The function also checks against making the node as attached directly to the technical root of the tree, thereby making sure there is just one root node at any instance.

Algorithm 1 misdirectedDependency()

Input: Node x

1: **if** $x.upos$ == "CCONJ" **and** $x.udeprel$ == "cc" **and** $x.parent.id < x.id$ **then**
2: **return true**
3: **end if**
4: **return false**

Algorithm 2 setParent()

Input: Node x, Original Parent y, New Parent Candidate z

1: **if** x will be attached non-projectively to z **or** z is *ROOT* **then**
2: $x.parent \leftarrow y$
3: **return false**
4: **else**
5: $x.parent \leftarrow z$
6: **return true**
7: **end if**

Having defined our wrapper functions, we start by trying to attach the conjunction at the same level to other siblings in Algorithm 3. We first check to see if there is a single remaining sibling that does not have a POS tag of `X`, `PUNCT`, or `SYM` since we want to avoid the linking of the conjunction to these POS tags. We do these checks in lines 4-15, and in case of a single sibling being available, attach the node therein, and return a **true** value.

In case the condition is not met, we try to find the nearest sibling that has the deprel as `conj`, and try attaching the conjunction there. Again, this condition might fail. As a last resort, we try to find indirect dependents of the conjunct the conjunction is attached to, and link the conjunction there. We limit the set of indirect dependents by restricting the deprels to `obl`, `xcomp`, `nmod`, and `nsubj`.

Notice how we decide on whether or not the algorithm terminates by continuously checking the condition of projectivity, and returning a value from the function only if the condition of projectivity with respect to the new parent is maintained. It is also important to note that we always limit our search for a suitable candidate to cases where the candidate occurs later than the conjunction we are trying to rehang.

Algorithm 3 attachToSibling()

Input: *node* such that $misdirectedDependency(node) ==$ **true**

{Try to attach to a sibling node}
$count \leftarrow 0$
$origParent \leftarrow node.parent$
for all *siblings* of *node* **do**
  if *siblings.upos* not in [“X”, “$PUNCT$”, “SYM”] **then**
    $TargetSibling \leftarrow siblings$
    $count \leftarrow count + 1$
  end if
end for
if $count == 1$ **then**
  {Just one sibling, attach to this sibling}
  if $setParent(node, origParent, TargetSibling)$ **then**
    return true
  end if
end if
{More than one siblings, narrow search by deprels}
for *sibling* of *node* **do**
  if *sibling.udeprel* == “*conj*” **and** *sibling.id* > *node.id* **then**
    if $setParent(node, origParent, sibling)$ **then**
      return true
    end if
  end if
end for
for *sibling* of *node* **do**
  if *sibling.udeprel* in [“*obl*”, “*xcomp*”, “*nmod*”, “*nsubj*”] **and** *node.id* < *sibling.id* **then**
    if $setParent(node, origParent, sibling)$ **then**
      return true
    end if
  end if
end for
return false

If there is no suitable candidate in the same level as the current level of the conjunction, we try to ascend one level and try to attach the node to the next aunt (parent's sibling) in Algorithm 4. We do not set any checks with respect to deprels, but still keep a check on the condition of projectivity and the node order.

Algorithm 4 attachToAunt()

Input: *node* such that $misdirectedDependency(node) ==$ **true**

{Try to attach to the first relevant aunt node}
$origParent \leftarrow node.parent$
$aunts = []$
for *sibling* of *origParent* **do**
  if $sibling.id > node.id$ **then**
    $aunts$.append($sibling$)
  end if
end for
if *aunts* is not empty **then**
  $setParent(node, origParent, aunts[0])$
end if
return false

In the event that a suitable candidate is not found, a **false** value is returned. This implies that our search for a suitable candidate has failed even after trying to ascend one level. As last resort, we try to attach the conjunction to the grandparent, while preserving projectivity in Algorithm 5.

Algorithm 5 attachToGrandparent()

Input: *node* such that $misdirectedDependency(node) ==$ **true**

{Try to attach to the grandparent node}
$origParent \leftarrow node.parent$
$grandparent \leftarrow origParent.parent$
if $setParent(node, origParent, grandparent)$ **then**
  return true
end if
return false

Having established all the possible cases, we can wrap them all in a nice function that takes care of all the cases, in priority order. The final algorithm is as given below:

Algorithm 6 fix_conj_head()

Input: *node* such that $misdirectedDependency(node) ==$ **true**

if $attachToSibling(node)$ **then**
  return
else if $attachToAunt(node)$ **then**
  return
else if $attachToGrandparent(node)$ **then**
  return
else
  Do Nothing
end if

Evaluation and Results

We implement the algorithm in form of a Udapi-python block. The runtime of the block for the data is as mentioned in Table 2, as run on Ubuntu 18.04 (64-bit) on a 4-core Intel i5-6300 HQ processor.

Language	RunTime (in ms)
af	68
ar	162

Table 2: Runtime for Algorithm with Udapy Python Block

After applying the algorithm on the data, there were 17 (0.92 % of identified misdirected instances), and 359 (25.44 % of identified misdirected instances) cases in af and ar data respectively, which could not be handled. With respect to the unhandled cases in both the datasets, the algorithm is designed to work in a way that it does not over-generate. As such, all the cases where the algorithm would have over-generated were not handled by the algorithm.

We hypothesized earlier that if the rehanging of the node requires a change in more than one level (of the level of wrong conjunct), it is likely to be an annotation error that needs manual correction. We found that to be true for more than 50% of the cases in either treebank with respect to all the unhandled cases.

For the evaluation, we randomly sample 100 instances identified with wrong dependencies before the correction from each treebank, and report the score on the counts of wrong dependencies before, and after the correction algorithm. The scores can be seen in Table 3.

Language	Corrected Instances
af	95
ar	97

Table 3: Results of Experiment on conj_head, evaluated with 100 random samples

With the defined algorithm in previous subsection, we were able to correct around a significant amount of flagged error cases, just by identifying the direction of dependency. Even with a consideration of 5% error, the algorithm is able to fix the dependencies effectively.

The algorithm was further tested on grc-PROIEL and grc-Perseus data, and the UDPipe parsers trained on the corrected data. The LAS scores when the parsers were tested on itself and on the other treebank, are as shown in Table below.

grc	PROIEL	Perseus
PROIEL	75.84	32.02
Perseus	43.23	70.54

Table 4: LAS, Before correction

grc	PROIEL	Perseus
PROIEL	77.86	32.14
Perseus	43.55	70.70

Table 5: LAS, After correction

Although there is not a significant change in LAS scores before and after the correction, the general increase in the scores can be attributed to the increased uniformity of the directions of association.