Splitting EUD graphs into trees: A quick and clatty approach

Mark Anderson Carlos Gómez-Rodríguez
Universidade da Coruña, CITIC
FASTPARSE Lab, LyS Research Group,
Departamento de Ciencias de la Computación y Tecnologías de la Información
{m.anderson,carlos.gomez}@udc.es

Abstract

We present the system submission from the FASTPARSE team for the EUD Shared Task at IWPT 2021. We engaged in the task last year by focusing on efficiency. This year we have focused on experimenting with new ideas on a limited time budget. Our system is based on splitting the EUD graph into several trees, based on linguistic criteria. We predict these trees using a sequence-labelling parser and combine them into an EUD graph. The results were relatively poor, although not a total disaster and could probably be improved with some polishing of the system’s rough edges.

1 Introduction

In our group’s submission to the IWPT 2020 shared task on EUD parsing (Dehouck et al., 2020), we focused on efficiency by applying distillation and training set reduction together with a rule-based approach to convert EUD graphs to UD trees that could be processed by an off-the-shelf parser. Here we describe our entry to the 2021 edition, where we keep the focus on algorithmic simplification of graphs, as well as a prioritisation of efficiency over raw accuracy, but we take the chance to explore different questions that we deem interesting in the context of a breadth-first exploration of the search space of parsing techniques, even if they are not (at least in their current form) competitive in terms of pushing speed or accuracy metrics.

In particular, we wanted to experiment with the application of sequence labelling parsing (Strzyz et al., 2019b) to the problem, which we apply to graph parsing for the first time. And more in particular, with the use of a linguistics-oriented approach (à la Dehouck et al. (2020)) to guide the parsing process by splitting the EUD graphs into coherent components that can be then parsed by a multitask learning system.

Sequence labelling, the task of assigning one discrete label to each token of a sequence, has long been used for various natural language processing tasks whose output can naturally be represented in this form, such as PoS tagging or named entity recognition. In the case of syntactic parsing, sequence labelling can be applied after defining an encoding that casts each possible syntactic tree for a sentence of length $n$ as a sequence of $n$ labels. While an early attempt to apply it to dependency parsing (Spoustová and Spousta, 2010) yielded subpar accuracy, the advances in machine learning architectures in the last decade have made this kind of approaches practically viable both for constituency (Gómez-Rodríguez and Vilares, 2018) and dependency parsing (Strzyz et al., 2019b). However, to our knowledge, sequence labelling approaches have not previously been tried for any sort of graph parsing.

One possible way of extending the search space of a parsing approach is to apply the approach to parse a constant amount of subgraphs (typically, two) whose union provides the final output. This has been applied to go beyond noncrossing dependency trees in transition-based dependency parsing by splitting trees into two subsets of arcs (planes) such that there cannot be crossings within each of them, but their union (the final output) can have crossing arcs (Gómez-Rodríguez and Nivre, 2010; Gómez-Rodríguez and Nivre, 2013; Fernández-González and Gómez-Rodríguez, 2018). In semantic parsing, it has also be used to extend the search space from noncrossing graphs to pagename-2 graphs by Sun et al. (2017), who use graph-based parsing to obtain two noncrossing graphs that are combined by Lagrangian relaxation. In the context of sequence labeling, this approach was recently applied by Strzyz et al. (2020) with similar goals and methods as the transition-based parsers above.
While all these approaches split the output with the goal of relaxing noncrossing constraints, the same can be applied to relax single-head constraints, i.e., go from tree to graph parsing. For example, any graph with in-degree at most 2 can trivially be expressed as the union of two trees. Here we apply that idea in the context of sequence labeling parsing, i.e., we try to generate several sequences of labels (via multitask learning), each of which represents a tree, and which together form an EUD graph.

However, all these splitting approaches share an underlying question: which of the (exponentially many) possible splits is more adequate for the model to adequately learn the parsing problem? The work cited above applies purely algorithmic criteria to choose a canonical split: lazy criteria to minimise the number of plane (subset) switches (Gómez-Rodríguez and Nivre, 2010) or the number of arcs assigned to the second plane (Strzyz et al., 2020), or systematic algorithms that assign crossing arcs to alternating planes (Sun et al., 2017). These provide splits that are related to the full parse by systematic structural criteria, but not by linguistic criteria. Since it has been repeatedly shown that it is possible to jointly learn different kinds of dependencies in such a way that they complement each other (e.g., with syntactic and semantic dependencies, as in Henderson et al., 2013; Zhou et al., 2020)) and sequence labeling parsing can benefit from integrating several linguistic representations using multitask learning (Strzyz et al., 2019a), what if we try to split parses in a linguistically meaningful way, yielding subsets of dependencies with a distinct meaning that can then be jointly learned? Here we evaluate such an approach.

2 Splitting graphs

The vast majority of nodes in a EUD graph only has one incoming edge. If we were only to use one edge per node, we would cover 94.15% of the edges. Only allowing a maximum of two incoming edges covers 99.53% edges; three covers 99.88%, and four covers 99.95%. Figure 2 shows how many nodes have different numbers of incoming edges. Note the logarithmic scale used for the number of nodes. We believe that our graph splitting process results in covering a maximum of two edges, although we have not checked it formally.

We attempted to split the trees in a linguistically grounded way. We first create what call the basic tree which most closely corresponds to the relative UD tree. We then create a relative, control, and conjunct tree. It is worth noting that, contrary to the work cited in the introduction where parses are split into two disjoint subgraphs, here we have four trees and all these trees can (and usually) overlap. We now describe the different trees into which we split the graph, as well as the collating procedure to combine output trees again into an EUD graph.

**Basic tree** This tree is made up of the EUD edges that correspond directly to the UD edges. With one exception for case marking. We add a relative position for the lemma (rather than the lemma itself). This means multi-word case marking is not covered. If no such edge exists we set

```
Figure 1: Basic tree split. Only ACL:TO changes to ACL:-1 for the relative lemma encoding.
```

```
Figure 2: Counts of nodes with x number of incoming edges.
```

the edge to (0, root). Although there should be no
way this introduces cycles, we check for them any-
where and if any are found we use the Chu-Liu-
Edmonds algorithm (CLE) (Chu and Liu, 1965;
Edmonds, 1967), setting scores for expected edges
to a sufficiently high value so that they are pri-
oritised, while the others are set very low. If the
MST tree has a different edge than in the basic
tree, we set that edge to (0, root). If the CLE al-
gorithm changes the ref edge, we change the in-
coming edge to its head to (0, root). An example
is shown in Figure 1.

Relative clause tree We take the basic tree of a
graph and replace incoming edges to nodes with ref edges. Again we check for cycles. This
Tree type was based on an error. We thought that
the relative pronoun had two incoming edges: one
from the head of the relative clause and one from
the referent. This meant we unnecessarily split
the basic and relative trees. An example of this
is shown in Figure 3.

Conjunct tree We start from the basic tree.
When an edge is “conj” we replace it with the edge
in the EUD column that has the same rel as the
conj head edge. We use the same cycle check as
for the previous trees. An example is shown in
Figure 4.

Control tree We take the basic tree again and
this time replace the original nsubj edges of a node
when its head has an incoming xcomp or ccomp
ege with the other nsubj edge in the EUD graph
for the node. We handle potential cycles as usual.
An example is shown in Figure 5. Another er-
ror is introduced here, where we don’t swap in the
ccomp edges.

Cycles Only Arabic-PADT has issues with
acyclicity after running CLE. So we just collapse
the edges that have been changed (this accounts
for three instances in the training data).

Collating trees into graphs As we operated on
a limited time budget, the collating method is egre-
giously simple. For each node, we take the set of
unique edges from all the predicted edges across
all trees. When an edge exists between \( w_i \) and \( w_j \)
in more than one tree, we use the label from the
first occurrence which is typically that of the basic
tree.

Table 1 shows the EULAS and ELAS when
splitting the gold graphs and collating them again.

We clearly can cover most of the graph edges with
this procedure with Arabic enhanced labels being
very low. We believe that this is a bug, but it could
be due to some inherent unexpected characteristic
of our basic splitting procedure.

<table>
<thead>
<tr>
<th>Treebank</th>
<th>EULAS</th>
<th>ELAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar-padtt</td>
<td>94.04</td>
<td>81.91</td>
</tr>
<tr>
<td>bg-btb</td>
<td>97.09</td>
<td>97.06</td>
</tr>
<tr>
<td>cs-cac</td>
<td>94.72</td>
<td>93.18</td>
</tr>
<tr>
<td>cs-fictree</td>
<td>94.21</td>
<td>91.75</td>
</tr>
<tr>
<td>cs-pdt</td>
<td>94.41</td>
<td>92.36</td>
</tr>
<tr>
<td>en-ewt</td>
<td>97.44</td>
<td>97.44</td>
</tr>
<tr>
<td>en-gum</td>
<td>97.09</td>
<td>97.09</td>
</tr>
<tr>
<td>et-edt</td>
<td>95.61</td>
<td>92.35</td>
</tr>
<tr>
<td>et-ewt</td>
<td>95.75</td>
<td>91.27</td>
</tr>
<tr>
<td>fi-ttd</td>
<td>92.73</td>
<td>87.13</td>
</tr>
<tr>
<td>fr-sequoia</td>
<td>96.22</td>
<td>96.22</td>
</tr>
<tr>
<td>it-isdt</td>
<td>96.32</td>
<td>95.98</td>
</tr>
<tr>
<td>lt-alksnis</td>
<td>94.08</td>
<td>87.35</td>
</tr>
<tr>
<td>lv-lvtrb</td>
<td>93.77</td>
<td>93.77</td>
</tr>
<tr>
<td>nl-alpino</td>
<td>98.07</td>
<td>98.01</td>
</tr>
<tr>
<td>nl-lassign</td>
<td>97.34</td>
<td>97.30</td>
</tr>
<tr>
<td>pl-lfg</td>
<td>99.02</td>
<td>99.02</td>
</tr>
<tr>
<td>pl-pdb</td>
<td>96.37</td>
<td>96.19</td>
</tr>
<tr>
<td>ru-syntagrus</td>
<td>97.97</td>
<td>97.68</td>
</tr>
<tr>
<td>sk-snk</td>
<td>96.23</td>
<td>94.18</td>
</tr>
<tr>
<td>sv-talbanken</td>
<td>96.31</td>
<td>96.31</td>
</tr>
<tr>
<td>ta-ttb</td>
<td>97.62</td>
<td>93.39</td>
</tr>
<tr>
<td>uk-iu</td>
<td>96.35</td>
<td>95.97</td>
</tr>
</tbody>
</table>

Table 1: Graphs formed from splitting the gold an-
notated development trees and subsequently collating
them again.

3 Parser

We use a BiLSTM network which has word
and character embeddings as input. We use a
sequence-labelling parser so the edges are pre-
dicted as separate labels for each token. Simi-
larly, the edge labels are predicted separately. But
both label predictions are jointly trained with a
hard-sharing multi-task architecture with equal
weighted loss contributions. UDPipe 2.0 was
used for tokenization, lemmatization, and tagging.
FastText word embeddings were used but limit vo-
cab space to 50k tokens for memory constraints
(Bojanowski et al., 2017). We then train 4 parsers
for each treebank which are trained on the data
generated by splitting the graph, i.e. there is one
parser for the basic trees, one for the relative trees,
and so on. Then the parsers are used to predict
their respective tree type and these are all collated
to create the predicted graphs for each treebank.

Sequence labelling parser (SEQLAB) is a pars-
ing approach based on encoding trees as a se-
They look like they were doberman pinchers who were shrunk.

Figure 3: Relative tree split. Top full EUD graph, middle basic tree, bottom relative tree.

We choose to use the original bracketing encoding from Strzyz et al. (2019b), as it does not require UPOS tags on decoding (the other leading encoding does). While there is a more recent bracketing encoding that covers more non-projectivity (Strzyz et al., 2020), this also involves splitting trees which we assumed would add too much complexity on top of our linguistic-based splitting. Our chosen encoding represents a tree as sequence of tags composed of left and right brackets representing each word’s incoming and outgoing arcs. Namely, the encoding for $w_i$ is based on:

1 $\epsilon_{ij} < i \land \epsilon_{ij} \in \mathcal{E}$

2 $\epsilon_{ij} \land j < i$

We repurposed a PyTorch biaffine implementation and edit it to be a simple sequence-labelling system, i.e. embedding layers, followed by a number of BiLSTM layers, and one MLP for predicting bracket tags and another for edge labels. The hyperparameters are shown in Table 2. The original code for the biaffine is no longer available but a similar version is still available. More details of the system can be found in Anderson and Gómez-Rodríguez (2021).

---

1 We use refactored encoding/decoding functions from https://github.com/mstrise/dep2label.

2 https://github.com/yzhangcs/parser
A clean and stable version is attached

Figure 4: Conjunct tree split. Top full EUD graph, middle basic tree, bottom conjunct tree.

I assume you have your answer and do not need me

Figure 5: Control tree split. Top full EUD graph, middle basic tree, bottom control tree.
Figure 6: The bracketing encoding introduced by Strzyz et al. (2019b).

Table 2: Network hyperparameters.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word embedding dimensions</td>
<td>300</td>
</tr>
<tr>
<td>Char embedding dimensions</td>
<td>100</td>
</tr>
<tr>
<td>Char BiLSTM dimensions</td>
<td>100</td>
</tr>
<tr>
<td>Embedding dropout</td>
<td>0.33</td>
</tr>
<tr>
<td>BiLSTM dimensions</td>
<td>500</td>
</tr>
<tr>
<td>BiLSTM layers</td>
<td>3</td>
</tr>
<tr>
<td>BiLSTM dropout</td>
<td>0.33</td>
</tr>
<tr>
<td>MLP layers</td>
<td>1</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.02</td>
</tr>
<tr>
<td>MLP dropout</td>
<td>0.33</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.9</td>
</tr>
<tr>
<td>L2 norm λ</td>
<td>0.9</td>
</tr>
<tr>
<td>Annealing</td>
<td>0.75&lt;sup&gt;∧&lt;/sup&gt;(t/5000)</td>
</tr>
<tr>
<td>ε</td>
<td>1×10&lt;sup&gt;-12&lt;/sup&gt;</td>
</tr>
<tr>
<td>Optimiser</td>
<td>Adam</td>
</tr>
<tr>
<td>Loss function</td>
<td>cross entropy</td>
</tr>
<tr>
<td>Epochs</td>
<td>200</td>
</tr>
<tr>
<td>Min vocab freq.</td>
<td>1</td>
</tr>
<tr>
<td>Batch size</td>
<td>32</td>
</tr>
<tr>
<td>Patience</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3: Average scores over all treebanks.

<table>
<thead>
<tr>
<th></th>
<th>UAS</th>
<th>ELAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUAS</td>
<td>86.66</td>
<td></td>
</tr>
<tr>
<td>EULAS</td>
<td>72.02</td>
<td></td>
</tr>
<tr>
<td>ELAS</td>
<td>69.21</td>
<td></td>
</tr>
</tbody>
</table>

4 Results

The results were rather underwhelming, but our system wasn’t an abject failure. Figure 7 shows the average performance of the parsers trained on each tree type. The performance is pretty stable across each type which is not surprising as the overall structure doesn’t vary greatly. But the average performance on the collated trees is quite a bit less as shown in Table 3. We decided to include EUAS which measures the unlabelled graph structure. This shows that the parser does learn the graph structure fairly well, but really struggles with labelling the edges. This could be due to appending the labels with the relative positioning of lemmas used for case marking making it harder to predict even the basic label type. Figure 8 shows the breakdown of the three metrics for each treebank. It is clear that for each treebank a fairly accurate prediction of the graph structure is achieved, but the labelled versions perform much worse. Table 4 shows the full results of out system on the test data. The performance across the board is fairly weak and resulted in the worst system which submitted predictions for the full treebank set (and was second last overall).

5 Discussion and conclusion

The relative tree split was based on a mistake. We should have left the REF edges in the basic tree and added the NSUBJ label variant to the referent in the relative tree. The way it is implemented, we lose those edges. Despite this error, we can still reconstruct most of the edges in the graphs. Beyond this, we can’t capture higher-order edges with this method. We did try using a SWEEP tree, to capture certain 3rd degree edges. But it seemed as the parser struggled to make sensible predictions and subsequently time ran out before we could test this thoroughly.

The collator is very naive. A major issue is introducing extra dummy root edges due to the nature of the split. Another thing we could have tried would have been to collate edges from trees that are only associated with the specific phenomenon of a given tree (i.e. conjunct trees only propagating conjunct edges.)

Also, looking at the difference in performance between EUAS and ELAS, it seems the labelling is bad. And the difference between EULAS and ELAS suggests this isn’t just a matter of the case marking messing things up. However, the use of relative positional encoding of the case marking might make it harder to learn the labels. Although, the LAS for each tree type isn’t that low. So it
could potentially be an issue about the way the trees are collated. Perhaps a first step would be to separate the relative case marking from the relation labels and treat it as a separate task in the MTL system.

We have presented a simple technique that can easily be extended (and implemented better) but manages to predict relatively accurate unlabelled graphs. It also isn’t an utter failure when considering labelled edges, but it seems curious that the performance drops so much compared to the unlabelled performance.

Acknowledgments

This work has received funding from the European Research Council (ERC), under the European Union’s Horizon 2020 research and innovation programme (FASTPARSE, grant agreement No 714150), from ERDF/MICINN-AEI (ANSWER-ASAP, TIN2017-85160-C2-1-R), from Xunta de Galicia (ED431C 2020/11), and from Centro de Investigación de Galicia “CITIC”, funded by Xunta de Galicia and the European Union (ERDF - Galicia 2014-2020 Program), by grant ED431G 2019/01.

References


Yoeng-Jin Chu and Tseng-Hong Liu. 1965. On the


