AMR Dependency Parsing with a Typed Semantic Algebra Supplementary Materials

Jonas Groschwitz[*†] **Matthias Lindemann**[*] **Meaghan Fowlie**[*]
Mark Johnson[†] **Alexander Koller**[*]
[*] Saarland University, Saarbrücken, Germany [†] Macquarie University, Sydney, Australia
jonasg|mlinde|mfowlie|koller@coli.uni-saarland.de
mark.johnson@mq.edu.au

A NP-completeness of the decoding problem

We prove NP-completeness for the well-typed decoding problem by reduction from HAMILTONIAN-PATH.

Let $G = (V, E)$ be a directed graph with nodes $V = \{1, \ldots, n\}$ and edges $E \subseteq V \times V$. A Hamiltonian path in G is a sequence $(v_1, \ldots, v_n)$ that contains each node of V exactly once, such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq n-1$. We assume w.l.o.g. that $v_n = n$. Deciding whether G has a Hamiltonian path is NP-complete.

Given G, we construct an instance of the decoding problem for the sentence $w = 1 \ldots n$ as follows. We assume that the first graph fragments shown in Fig. 1a (with node label "i") is the only graph fragment the supertagger allows for 1, …, $n-1$, and the second one (with node label "f") is the only graph fragment allowed for n. We let $\omega(i \to k) = 1$ if $(i, k) \in E$, and zero otherwise.

Under this construction, every well-typed AM dependency tree for w corresponds to a linear sequence of nodes connected by edges with label APP$_s$ (see Fig. 1c for an example) More specifically, n is a leaf, and every node except for n has precisely one outgoing APP$_s$ edge; this is enforced by the well-typedness. Because of the edge scores, the score of such a dependency tree is $n-1$ iff it only uses edges that also exist in G; otherwise the score is less than $n-1$. Therefore, we can decide whether G has a Hamiltonian path by running the decoder, i.e. computing the highest-scoring well-typed AM dependency tree t for w, and checking whether the score of t is $n-1$.

B Neural Network Details

We implemented the supertagger (Section 5.1) and the local dependency model (Section 5.3) in PyTorch, and used the original DyNet implementation

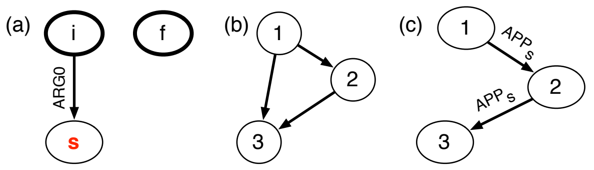

Figure 1: (a) The two graph fragments required for the NP-completeness proof. (b) An example graph and (c) the AM dependency tree corresponding to its Hamiltonian path.

of (Kiperwasser and Goldberg, 2016) (short K&G) for the K&G model. Further details are:

1. As pre-trained embeddings, we use GloVE (Pennington et al., 2014). The vectors we use have 200 dimensions and are trained on Wikipedia and Gigaword. We add randomly initialized vectors for the `name`, `date` and `number` tokens and for the unknown word token (if no GloVE vector exists). We keep these embeddings fixed and do not train them.

2. For the learned word embeddings, we follow K&G in all our models in using a word dropout of $\alpha = 0.25$. That is, during training, for a word that occurs k times in the training data, with probability $\frac{\alpha}{k+\alpha}$ we instead use the word embedding for the unknown word token instead of w_i.

3. The character-based encodings c_i for the supertagger are generated by a single layer LSTM with 100 hidden dimensions, reading the word left to right. If a word (or sequence of words) is replaced by e.g. a `name` token during pre-processing, the character-based encoding reads the original string instead (this helps to classify names correctly as country, person etc.).

4. To prevent overfitting, we add dropout of 0.5

Optimizer	Adam
Learning Rate	0.004
Epochs	37
Pre-trained word embeddings	glove.6B
Pre-trained word emb. dimension	200
Learned word emb. dimension	100
POS embedding dimension	32
Character encoding dimension	100
α (word dropout)	0.25
Bi-LSTM layers	2 (stacked)
Hidden dimensions in each LSTM	256
Hidden units in MLPs	256
Internal dropout of LSTMs, MLPs	0.5
Input vector dropout	0.8

Table 1: Hyperparameters used for training the supertagger (Section 5.1)

Optimizer	Adam
Learning rate	default
Epochs	16
Word embedding dimension	100
POS embedding dimension	20
Type embedding dimension	32
α (word dropout)	0.25
Bi-LSTM layers	2 (stacked)
Hidden dimensions in each LSTM	128
δ	0.2
Hidden units in MLPs	100

Table 2: Hyperparameters used for training K&Gś model (Section 5.2)

in the LSTM layers of all the models except for the K&G model which we keep as implemented by the authors. We also add 0.5 dropout to the MLPs in the supertagger and local dependency model.

5. For the K&G model with the fixed-tree decoder, we perform early stopping computing the Smatch score on the development set with 2 best supertags after each epoch.

6. Hyperparameters for the different neural models are detailed in Tables 1, 2 and 3. We did not observe any improvements when increasing the number of LSTM dimensions of the K&G model.

Optimizer	Adam
Learning Rate	0.004
Epochs	35
Pre-trained word embeddings	glove.6B
Pre-trained word emb. dimension	200
POS embedding dimension	25
Bi-LSTM layers	2 (stacked)
Hidden dimensions in each LSTM	256
Hidden units in MLPs	256
Internal dropout of LSTMs, MLPs	0.5
Input vector dropout	0.8

Table 3: Hyperparameters used for training the simplified dependency model (Section 5.3)

C Decoding Details

The goal item of the decoders is one with empty type that covers the complete sentence. In practice, the projective decoder always found such a derivation. However, in in a few cases, this cannot be achieved by the fixed-tree decoder with the given supertags. Thus, we take instead the item which minimizes the number of open sources in the resulting graph.

When the fixed-tree decoder takes longer than 20 minutes using k best supertags, it is re-run with $k-1$ best supertags. If $k=0$, a dummy graph is used instead. Typically, the limit of 20 minutes is exceeded one or more times by the same sentence of the test set.

With the projective decoder, in most runs, 1 or 2 sentences took too long to parse and we used a dummy graph instead.

We trained the supertagger and all models 4 times with different initializations. For evaluation, we paired each edge model with a supertag model such that every run used a different edge model and different supertags. The reported confidence intervals are 95% confidence intervals according to the t-distribution.

D Pre- and postprocessing Details

D.1 Aligner

We use a heuristic process to generate alignments satisfying the conditions above. Its core principles are similar to the JAMR aligner of (Flanigan et al., 2014). There are two types of actions:

Action 1: Align a word to a node (based on the word and the node label, using lexical similarity, handwritten rules[1] and WordNet neighbourhood; we align some name and date patterns directly). That node becomes the lexical node of the alignment.

Action 2: Extend an existing alignment to an adjacent node, such as from “write” to “person” in the example graph in the main paper. Such an extension is chosen on a heuristic based on

1. the direction and label of the edge along which the alignment is split,

2. the labels of both the node we spread from, and the node we spread to, and

3. the word of the alignment.

We disallow this action if the resulting alignment would violate the single-root constraint of Section 4.2 in the main paper.

Each action has a basic heuristic score, which we increase if a nearby node is already aligned to a nearby word, and decrease if other potential operations conflict with this one. We remove We iteratively execute the highest scoring action until all heuristic options are exhausted or all nodes aligned. We then align remaining unaligned nodes to words near adjacent alignments.

D.2 Postprocessing

Having obtained an AM dependency tree, we can recover an AM term and evaluate it. During postprocessing we have to re-lexicalize the resulting graph according to the input string. For relatively frequent words in the training data (occurring at least 10 times), we take the supertagger’s prediction for the label. For rarer words, the neural label prediction accuracy drops, and we simply take the node label observed most often with the word in the training data. For unseen words, if the lexicalized node has outgoing ARGx edges, we first try to find a verb lemma for the word in WordNet (Miller, 1995) (we use version 3.0). If that fails, we try, again in WordNet, to find the closest verb derivationally related to any lemma of the word. If that also fails, we take the word literally. In any case, we add “-01” to the label. If the lexicalized node does not have an outgoing ARGx edge, we try fo find a noun lemma for the word in Wordnet, and otherwise take the word literally.

[1]E.g. the node label “have-condition-91” can be aligned to “if” and “otherwise”.

For names, we again simply look up name nodes and wiki entries observed for the word in the training data, and for unseen names use the literal tokens as the name and no wiki entry. We recover dates and numbers straightforwardly.

References

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations. *Transactions of the Association for Computational Linguistics* 4:313–327.

George A Miller. 1995. Wordnet: a lexical database for english. *Communications of the ACM* 38(11):39–41.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*.