# UniParse: A universal graph-based parsing toolkit

**Daniel Varab**
IT University
Copenhagen, Denmark
djam@itu.dk

**Natalie Schluter**
IT University
Copenhagen, Denmark
natschluter@itu.dk

## Abstract

This paper describes the design and use of the graph-based parsing framework and toolkit UniParse, released as an open-source python software package developed at the IT University, Copenhagen Denmark. UniParse as a framework novelly streamlines research prototyping, development and evaluation of graph-based dependency parsing architectures. The system does this by enabling highly efficient, sufficiently independent, readable, and easily extensible implementations for all dependency parser components. We distribute the toolkit with ready-made pre-configured re-implementations of recent state-of-the-art first-order graph-based parsers, including highly efficient Cython implementations of feature encoders and decoding algorithms, as well as off-the-shelf functions for computing loss from graph scores.

## 1 Introduction

**Motivation.** While graph-based dependency parsers are theoretically simple models, extensible and modular implementations for sustainable parser research and development have to date been severely lacking in the research community. Contributions to parsing research generally centres around particular components of parsers in isolation, such as novel decoding algorithms, novel arc encodings, or novel learning architectures. However, due to perceived gains in performance or due to the lack of foresight in writing sustainable code, these components are rarely implemented modularly or with extensibility in mind. This applies to prior sparse-feature dependency parsers (McDonald and Pereira (2006)'s MST parser), as well as recent state-of-the-art neural parsers (Kiperwasser and Goldberg, 2016; Dozat and Manning, 2017). Implementations of parser components are generally tightly coupled to one another which heavily hinders their usefulness in future research.

With UniParse, we provide a flexible, highly expressive, scientific framework for easy, low-barrier of entry, highly modular, efficient development and fair benchmarking of graph-based dependency parsing architectures. With the framework we distribute pre-configured state-of-the-art first-order sparse and neural graph-based parser implementations to provide strong baselines for future research on graph based dependency parsers.

**Novel contributions**

- We align sparse feature and neural research in graph-based dependency parsing to a **common terminology**. With this shared terminology we develop a unified framework for the UniParse toolkit to rapidly prototype new parsers and easily compare performance to previous work.
- Prototyping is now rapid due to **modularity**: parser components may now be developed in isolation, with no resulting loss in efficiency. Measuring the empirical performance of a new decoder no longer require implementing an encoder, and investigating the synergy between a learning strategy and a decoder no longer requires more than a flag or calling a library function.
- **Preprocessing is now made explicit** within its own component and is thereby adequately isolated and portable.
- **The evaluation module is now easy to read and fully specified.** We specify the subtle differences in computing unlabeled and labeled arc scores (UAS, LAS) from previous



*Proceedings of the 22nd Nordic Conference on Computational Linguistics (NoDaLiDa)*, pages 406–410
Turku, Finland, 30 September – 2 October, 

literature and have implemented these in UniParse in an explicit manner.

- To the best of our knowledge, UniParse is the first attempt at **unifying existing dependency parsers to the same code base**. Moreover, UniParse is to our knowledge the first attempt to enable first-order sparse-feature dependency parsing in a *shared* python codebase.

We make the parser freely available under a GNU General Public License[1].

## 2 Terminology of a unified dependency parser

Traditionally, a graph-based dependency parser consists of three components. An *encoder* $\Gamma$, a set of *parameters* $\lambda$, and a *decoder* $h$. The possible dependency relations between all words of a sentence $S$ are modeled as a complete directed graph $G_S$ where words are nodes and arcs are the relations. An arc in $G_S$ is called a factor which $\Gamma$ associates with a $d$-dimensional feature vector, its *encoding*. The set of parameters $\lambda$ are then used to produce scores from the constructed feature vectors according to some learning procedure. These parameters are optimized over treebanks. Lastly a decoder $h$ is some maximum spanning tree algorithm with input $G_S$ and scores for factors of $G_S$ given by $\lambda$; it outputs a well-formed dependency tree, which is the raw output of a dependency model.

Recent work on neural dependency parsers learns factor embeddings discriminatively alongside the parameters used for scoring. The result is that $\Gamma$ and $\lambda$ of dependency parsers fuse together into a union of parameters. Thus, in this work we fold the notion of encoding into the parameter space. Now, for the neural models, all parameters are trainable, whereas for sparse-feature models, the encodings of sub-sets of arcs are non-trainable. So the unified terminology addresses only parameters $\lambda$ and a decoder $h$.

## 3 API and the joint model architecture

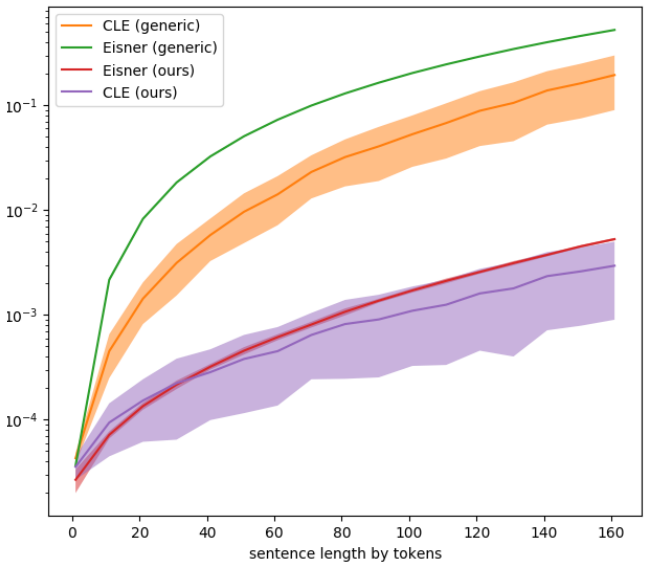
We provide two levels of abstraction for implementing graph-based dependency parsers. First, our descriptive high-level approach focuses on expressiveness, enabling models to be described in just a few lines of code by providing an interface where the required code is minimal, only a means to configure design choices. Second, as an alternative to the high-level abstraction we emphasise that parser definition is nothing more than a composition of pre-configured low-level modular implementations. With this we invite cherry picking of the included implementations of optimised decoders, data preprocessors, evaluation module and more. We now briefly overview the basic use of the joint API and list the central low-level module implementations included with the UniParse toolkit.

**Elementary usage (high level).** For ease of use we provide a high-level class to encapsulate neural training. Its use results in a significant reduction in the amount of code required to implement a parser and counters unwanted boilerplate code. It provides default configurations for all included components, while enabling custom implementation whenever needed. Custom implementations are only required to be callable and adheres to the framework's function definition. The minimum requirement with the use of this interface is a parameter configuration, loss function, optimizer, and batch strategy. In Listing 1 we show an example implementation of Kiperwasser and Goldberg (2016)'s neural parser in only a few lines. The full list of possible arguments along with their interfaces can be found in the toolkit documentation.

**Vocabulary.** This class facilitates preprocessing of CoNLL-U formatted files with support for out-of-vocabulary management and alignment with pre-trained word embeddings. Text preprocessing strategies have significant impact on NLP model performance. Despite this, little effort has put into describing such techniques in recent literature. Without these details preprocessing becomes yet another hyper-parameter of a model, and obfuscates research contribution. In the UniParse toolkit, we include a simple implementation for recently employed techniques in parsing for token cleaning and mapping.

**Batching.** UniParse provides functionality to organise tokens into batches for efficient computation and learning. We provide several configurable implementations for different batching strategies. This includes 1. batching by sentence length (bucketing), 2. fixed-length batching with padding, and 3. clustered-length batching as seen

[1] `github.com/danielvarab/uniparse`




```python
from uniparse import *
from uniparse.models import *

vocab = Vocabulary().fit(train)
params = KiperwasserGoldberg()
model = Model(params,
              decoder="eisner",
              loss="hinge",
              optimizer="adam",
              vocab=vocab)
metrics = model.train(train, dev,
                      epochs=30)
test_metrics = model.evaluate(test)
```

| Algorithm | en_ud | en_ptb | sents/s |
|---|---|---|---|
| Eisner (generic) | 96.35 | 479.1 | $\sim 80$ |
| Eisner (ours) | 1.49 | 6.31 | $\sim 6009$ |
| CLE (generic) | 19.12 | 93.8 | $\sim 404$ |
| CLE (ours) | 1.764 | 6.98 | $\sim 5436$ |

Figure 1: (Right code snippet) Implementation of Kiperwasser and Goldberg (2016)'s neural parser in only a few lines using UniParse.
(Right table and left figure) Number of seconds a decoder takes to decode an entire dataset. Score matrices are generated uniformly in the range $[0, 1]$. The random generated data has an impact on CLE due to its greedy nature; The figure demonstrates this by the increasingly broad standard deviation band. Experiments are run on an Ubuntu machine with an Intel Xeon E5-2660, 2.60GHz CPU.

in the codebase for Dozat and Manning (2017)[2] (this is not described in the published work). With unobtrusive design in mind any alternative custom batching strategy may be employed directly, no interaction with the framework is needed.

**Decoders** We include optimised Cython implementations of first-order decoder algorithms with the toolkit. This includes Eisner's algorithm (Eisner, 1996) and Chu-Liu-Edmonds (Chu and Liu, 1965; Edmonds, 1967; Zwick, 2013). In Figure 1 we compare the performance of our decoder implementations against pure python implementations[3] on randomised score input. Our implementations outperform a pure python implementations by a order of several magnitudes.

**Evaluation.** UAS and LAS are central dependency parser performance metrics, measuring unlabeled and labeled arc accuracy respectively with $\text{UAS} = \frac{\#\text{correct arcs}}{\#\text{arcs}}$ and $\text{LAS} = \frac{\#\text{correctly labeled arcs}}{\#\text{arcs}}$. Unfortunately, there are also a number unreported preprocessing choices preceding the application of these metrics, which renders direct comparison of parser performance in the literature futile, regardless of how well-motivated these preprocessing choices are. These are generally discovered by manually screening the code implementations when these implementations are made available to the research community. Two important variations found in state-of-the-art parser evaluation are the following.

1. **Punctuation removal.** Arcs incoming to any punctuation are somtimes removed for evaluation. Moreover, the definition of punctuation is not universally shared. We provide a clear python implementation for these metrics with and without punctuation arc deletion before application, where the definition of punctuation is clear: punctuation refers to tokens that consist of characters complying to the Unicode punctuation standard.[4] This is the strategy employed by the widely used Perl evaluation script, which to our knowledge, originates from the CoNLL 2006 and

[2] https://github.com/tdozat/Parser-v1

[3] https://github.com/LxMLS/lxmls-toolkit/blob/1bdc382e509d24b24f581c1e1d78728c9e739169/lxmls/parsing/dependency_decoder.py

[4] https://www.compart.com/en/unicode/category

| Parser configurations | Dataset | UAS wo.p.. original | LAS wo.p. original | UAS wo.p. | LAS wo.p. | UAS w.p. | LAS w.p. |
|---|---|---|---|---|---|---|---|
| Kiperwasser and Goldberg (2016) | en_ud | — | — | 87.71 | 84.83 | 86.80 | 85.12 |
| | en_ptb | 93.32 | 91.2 | 93.14 | 91.57 | 92.56 | 91.17 |
| | da | — | — | 83.72 | 79.49 | 83.24 | 79.62 |
| Dozat and Manning (2017) | en_ud | — | — | 91.47 | 89.38 | 90.74 | 89.01 |
| | en_ptb | 95.74 | 95.74 | 95.43 | 94.06 | 94.91 | 93.70 |
| | da | — | — | 87.84 | 84.99 | 87.42 | 84.98 |
| MSTparser (2006) + extensions | en_ud | — | — | 75.55 | 66.25 | 73.47 | 65.20 |
| | en_ptb | — | — | 76.07 | 64.67 | 74.00 | 63.60 |
| | da | — | — | 68.80 | 55.30 | 67.17 | 55.52 |

Table 1: UAS/LAS for included parser configurations. We provide results with (w.p.) and without (wo.p.) punctuation. For the English universal dependencies (UD) dataset we exclude the github repository suffix *EWT*. Regarding (Dozat and Manning, 2017), despite having access to the published TensorFlow code of we never observed scores exceed 95.58.

2007 shared tasks.[5] We infer this from references in (Buchholz and Marsi, 2006).

2. **Label prefixing.** Some arc labels are "composite", their components separated by a colon. An example from the English Universal Dependencies data set is the label `obl:tmod`. The official CoNLL 2017 shared-task evaluation script[6] allows partial matching of labels based on prefix matches for components, for example matching to `obl` of `obl:tmod` giving full points. We include this variant in the distributed UniParse evaluation module.

**Loss Functions.** Common loss functions apply to scalar values, or predictions vectors representing either real values or probabilities. However loss functions for dependency parsers are unorthodox in that they operate on graphs, which has been dealt with in various creative ways over the years. We include a set of functions that apply to first-order parser graphs which are represented as square matrices. In the future we hope to expand this set for first-order, as well as explore higher-order structures.

**Callbacks.** While we have done our uttermost to design UniParse in a unobtrusive manner, few limitations may occur when developing, and especially during exploration of model configurations when using the high-level model class. This could be the likes of manual updating of optimisers learning rates during training, or logging granulated loss and accuracy. To accommodate this we include callback functionality which hooks into the training procedure enabling users to do the last few things perhaps inhibited by the framework. We include a number of useful pre-implemented callback utilities, such as a Tensorboard logger[7], model saver, and a patience mechanism for early stopping.

**Included parsers.** We include three state-of-the-art first-order dependency parser implementations as example configurations of UniParse: McDonald and Pereira (2006)'s MST sparse-feature parser [8], Kiperwasser and Goldberg (2016) and Dozat and Manning (2017)'s graph-based neural parsers. Experiments are carried out on English and Danish: the Penn Treebank (Marcus et al., 1994) (en_ptb, training on sections 2-21, development on section 22 and testing on section 23), converted to dependency format following the default configuration of the Stanford Dependency Converter (version $\geq$ 3.5.2), and the English (en_ud), and Danish (da) datasets from Version 2.1 of the Universal Dependencies project (Nivre et al., 2017). Table 1 shows how our parser configurations perform compared with the originally reported parser performance.

[5] `https://depparse.uvt.nl/SoftwarePage.html#eval07.pl`

[6] `https://universaldependencies.org/conll17/baseline.html`

[7] `github.com/tensorflow/tensorboard`

[8] Note that this MST parser implementation consists of a restricted feature set and is only a first-order parser, as proof of concept.

## 4 Concluding remarks

In this paper, we have described the design and usage of UniParse, a high-level un-opinionated framework and toolkit that supports both feature-based models with on-line learning techniques, as well as recent neural architectures trained through backpropagation. We have presented the framework as answer to a long-standing need for highly efficient, easily extensible, and, most of all, directly comparable graph-based dependency parsing research.

The goal of UniParse is to ease development and evaluation of graph-based syntatic parsers. Future work includes extending UniParse to a general parsing pipeline from raw text.

## References


Sabine Buchholz and Erwin Marsi. 2006. Conll-x shared task on multilingual dependency parsing. In *In Proceedings of CoNLL*, pages 149–164. Association for Computational Linguistics.

Y.J. Chu and T.H. Liu. 1965. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14:13961400.

Timothy Dozat and Christopher M. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *Proceedings of ICLR*.

J. Edmonds. 1967. Optimum branchings. *J. Res. Nat. Bur. Standards*, 71B:233240.

Jason M. Eisner. 1996. https://doi.org/10.3115/992628.992688 Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, COLING '96, pages 340–345, Stroudsburg, PA, USA. Association for Computational Linguistics.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the ACL*, 4:313–327.

Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 114–119, Stroudsburg, PA, USA. Association for Computational Linguistics.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*. Association for Computational Linguistics.

Joakim Nivre et al. 2017. http://hdl.handle.net/11234/1-2515 Universal dependencies 2.1. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Uri Zwick. 2013. http://www.cs.tau.ac.il/ zwick/grad-algo-13/directed-mst.pdf Lecture notes on "analysis of algorithms": Directed minimum spanning trees.