PanParser: a Modular Implementation for Efficient Transition-Based Dependency Parsing

Lauriane Aufrant[1,2] **Guillaume Wisniewski**[1]

[1]LIMSI, CNRS, Univ. Paris-Sud, Université Paris-Saclay, 91 405 Orsay, France
[2]DGA, 60 boulevard du Général Martial Valin, 75 509 Paris, France
{lauriane.aufrant,guillaume.wisniewski}@limsi.fr

First version: March, 2016. Last update: April, 2017.

Abstract

We present PanParser, a Python framework for transition-based structured prediction that is primarily intended for dependency parsing, but has also applications in other tasks. The focus of PanParser is on algorithmic diversity and modularity; we believe it can be of use both as a state-of-the-art parser and for research on parsing. Its long-term purpose, which current version fulfills to some extent, is to unify all state-of-the-art methods under the same framework, for better comparison and to fill the gaps of unexplored variants in a painless but exhaustive way.[1]

1 Introduction

PanParser is not yet another implementation of a transition-based dependency parser. Transition-based dependency parsing has been an active field in the last few years, and several open source parsers have been released, each one implementing a new alternate paradigm. However, such algorithms have a large variability, with eg. various feature templates, transition systems, training strategies or classifiers, and the published parsers often adopt a specific variant, thus making fair benchmarking harder.

Hence, the purpose of the PanParser framework is to provide a modular architecture, in which most state-of-the-art parsing systems can be implemented, or extended, in a light and straightforward way. In addition to providing an easy way to train accurate models for parsing any language, it is then particularly valuable for research purpose and exhaustive experiments on parser design.

In this framework, a transition-based parser is considered as the association of several components, each one with several built-in versions:

- A *transition system*, associating parse trees with transition sequences: ArcEager, ArcHybrid, ArcStandard, NonMonotonicArcEager, variants for partial output and short-spanned dependencies, free choice of ROOT position
- A *classifier*, seen as a black box mapping feature representations to scored transitions: multiclass averaged perceptron, feedforward neural network, joint models
- A *search* component, which maps a model to transition sequences: greedy decoding, beam search
- An *oracle* component, which maps gold annotations to gold transitions: static oracle, dynamic oracle
- A *training strategy*, i.e. a choice of training configurations to update the model: local and global training, early update and max-violation with several variants, parameterized error exploration
- A *feature set*: delexicalized parsing, coarse or fine-grained tags, morphological features, free template design
- *Enriched input* management, with sparsity support at various levels: training on partially annotated data, prediction under partial constraints (when the head or dependency direction is already provided for a few tokens), training to leverage partial constraints, prediction of partial trees with dedicated training, fine-grained subsampling

[1]This document and the corresponding software are updated on a regular basis to include new functionalities. The latest version is available at https://perso.limsi.fr/aufrant.

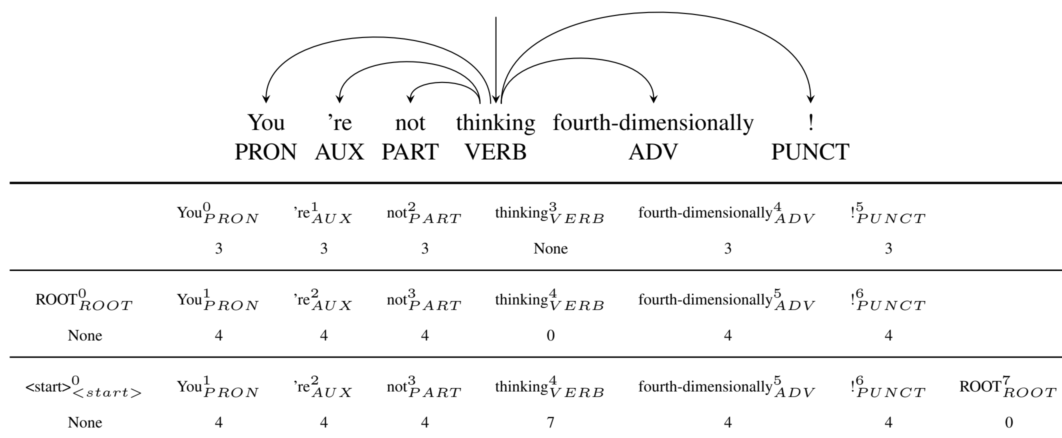
All those components are compatible, as they are interfaced under a unified architecture for structured prediction, that takes care of the main training and prediction logic, plus some extra evaluation utilities. This structured prediction core module is in fact not specific to parsing, and can also be used for other structured prediction tasks, as shown with the built-in PoS tagger. This framework has several other perks, like the ability to exploit non-projective data to train even projective parsers (see Appendix A), and extensive utilities for error analysis.

The whole is written in Python in a modular way, which makes it easy for the user to extend the built-in components with custom variants. For instance, adding the ARCHYBRID transition system (with full compatibility with all other components) was done in 150 lines of code.

Section 2 shows a straightforward use of PanParser to annotate new data. After a brief presentation of each built-in component and the corresponding literature (Section 3), more advanced functionalities are presented in Section 4 and technical details on the implementation are provided in Section 5.

2 Main usage

Parse representation Dependency parsing consists in analyzing the syntactic structure of a sentence by mapping it to a tree. Formally, a unique head token is assigned to each token of the sentence (apart from the root), avoiding cycles. To alleviate edge effects due to a different handling of the root of the sentence, a dummy ROOT token is generally inserted before or after the sentence tokens. Thus, a dependency parse is modeled by a list of head tokens with padding elements: Figure 1 illustrates the three options.



	You$^{0}_{PRON}$	’re$^{1}_{AUX}$	not$^{2}_{PART}$	thinking$^{3}_{VERB}$	fourth-dimensionally$^{4}_{ADV}$	!$^{5}_{PUNCT}$	
	3	3	3	None	3	3	
ROOT$^{0}_{ROOT}$	You$^{1}_{PRON}$	’re$^{2}_{AUX}$	not$^{3}_{PART}$	thinking$^{4}_{VERB}$	fourth-dimensionally$^{5}_{ADV}$	!$^{6}_{PUNCT}$	
None	4	4	4	0	4	4	
<start>$^{0}_{<start>}$	You$^{1}_{PRON}$	’re$^{2}_{AUX}$	not$^{3}_{PART}$	thinking$^{4}_{VERB}$	fourth-dimensionally$^{5}_{ADV}$	!$^{6}_{PUNCT}$	ROOT$^{7}_{ROOT}$
None	4	4	4	7	4	4	0

Figure 1: Parse representations with various ROOT positions.

Data format All data provided to PanParser must follow the Pan4 format, where each parse is a tuple of 4 lists: the sentence words (as strings), the PoS or morphological tags (as strings or dictionaries), the heads (the index of the head when specified, `None` otherwise) and the labels (as strings, or `None`). The same format is used for both training and prediction, in the latter case the heads and labels are simply ignored.

The parser can work internally with the dummy ROOT token either in first or last positions, but this is controlled by model parameters: the input format always supposes double padding (i.e. with ROOT at the end).

For CoNLL-X and CoNLL-U formats, the `read_conll` helper function can be used to load a treebank in this format. It has two filtering options: the `multiroots` parameter (default: `false`) can be set to prevent filtering of trees with multiple roots, which happens sometimes depending on the annotation scheme, and the `projective_only` parameter (default: `false`) removes non-projective sentences from the dataset.

```
from dependency_parser import read_conll
with open("en-train.conllu", "rt") as fh:
    dataset = read_conll(fh)
```

Parser usage Creating and using a new parser is done straightforwardly with three functions: `train_dependency_parser`, `process_dependency_parser` and `eval_dependency_parser`. For each, `listener=FrogressListener` can be used to improve the display with a progress bar, but this option requires the `frogress` external library.

```
# train a new parser with default parameters
from dependency_parser import train_dependency_parser
from misc.io import FrogressListener
parser = train_dependency_parser(None, dataset, n_epoch, listener=FrogressListener)
# annotate an input dataset and dump it in a CoNLL-U file
from dependency_parser import process_dependency_parser
from conll.io import wt, feat2col
with open("out.conllu", "wt") as out:
    wt(out, process_dependency_parser(parser, newdataset), preprocess=feat2col)
# print the parser UAS
from dependency_parser import eval_dependency_parser
print(eval_dependency_parser((parser, testset)))
```

The default is an unlabeled ArcEager beam parser, but this can be changed. The `train_dependency_parameter` procedure supports a number of additional parameters; see Section 4 and the in-code documentation for more details.

It is also possible to create a parser by using its class name directly, which gives additional control on the feature set (default: `zn11_features`, which corresponds to Zhang and Nivre (2011)'s templates augmented with transition history).

```
from dparser.features import zn11_features
parser = ArcEagerBeamParser(zn11_features)
train_dependency_parser(parser, dataset, n_epoch)
print(eval_dependency_parser((parser, testset)))
```

3 System components

Transition system In the transition-based approach, parsing is done in a shift-reduce manner, by a sequence of transitions (or actions), which incrementally build the parse tree.

PanParser is currently shipped with several transition systems. We only present here briefly the popular ARCEAGER (Nivre, 2004) system, and address the reader to Appendix A for full information on other supported transition systems.

The ARCEAGER system is projective, i.e. it forbids crossing edges by design, and stack-based: the parse configuration is given by a single stack (σ) of currently processed tokens, in addition to the buffer (β) of not yet processed tokens. Table 1 presents the preconditions and effects of its transitions.

Transition	Configuration		Result	Precondition
SHIFT	$(\sigma, \; b\vert\beta, \; P)$	$\Rightarrow$	$(\sigma\vert b, \; \beta, \; P)$	if b is a word
LEFT	$(\sigma\vert s, \; b\vert\beta, \; P)$	$\Rightarrow$	$(\sigma, \; b\vert\beta, \; P + (b \rightarrow s))$	if s is a word and s is unattached
RIGHT	$(\sigma\vert s, \; b\vert\beta, \; P)$	$\Rightarrow$	$(\sigma\vert s\vert b, \; \beta, \; P + (s \rightarrow b))$	
REDUCE	$(\sigma\vert s, \; \beta, \; P)$	$\Rightarrow$	$(\sigma, \; \beta, \; P)$	if s is attached

Table 1: ARCEAGER transition system.

Classifier In each parse configuration, the next transition to perform is predicted by a classifier. Following Collins and Roark (2004), we use a multi-class averaged perceptron: after training, the final parameters are obtained by averaging over values yielded after each perceptron update.

In PanParser, the classifier is considered as a black box: it can be replaced by any multi-class classifier as long as it respects the same predict/update API. There is support for joint prediction and work has been started to include feed-forward neural networks (already available for the built-in tagger, but the current neural parser still performs poorly and is not recommended).

Feature set There is a large body of research on feature engineering, i.e. the choice of which features to extract in a given configuration, and then feed to the classifier.

The first step is actual feature extraction. For transition-based parsing, it is typically based on the top of the stack and buffer: wordform of the first token in buffer, PoS tag of the (already attached) children of the stack head, etc. In addition to wordforms, morphological features and coarse or fine-grained PoS tags, pre-trained word embeddings can also be collected for any such token.

Then, using these features is straightforward for non-linear classifiers: they are simply provided to the classifier, which handles the rest. For linear classifiers, things get more complicated, because feature interactions must be handled at the feature level. Hence, this requires *feature templates* to select the feature tuples sent to the classifier. The templates of Zhang and Nivre (2011) are quite popular, since they include non-local information which boosts prediction a lot. PanParser includes those templates and uses them by default, but it is also possible to use easily any custom template, based on the extracted features.

Finally, some use cases like cross-lingual parsing require parser delexicalization. Instead of actually transforming the data, this can be handled at the feature level, by disabling all lexicalized feature templates, which PanParser also supports.

Search Since the search space is of exponential size, inexact search is employed, both for training and prediction. Greedy decoding is faster, but beam search (Zhang and Clark, 2008) yields more accurate parsers: transition sequences are scored by summing the scores of all their transitions, and at each step all possible actions are considered for each hypothesis in the beam, after which only the k-best resulting configurations are kept, for a fixed k value.

Training strategy In standard greedy training, a perceptron update is considered for each transition in the predicted sequence and it is performed whenever the followed transition is not in a pre-computed reference derivation (which may not be the unique derivation leading to the reference parse tree). With beam search this straightforward strategy is not applicable: according to Zhang and Nivre (2012), this learning strategy is local, while beam scores are global, and therefore yields poor performance. Instead, two main strategies have been proposed: EARLY UPDATE (Collins and Roark, 2004), in which decoding is stopped (and an update ensues) as soon as the reference has dropped out of the beam, and MAX-VIOLATION (Huang et al., 2012) that behaves the same, except that in case of an error, decoding is still pursued and the update occurs when the gap between one-best predicted and reference scores is the largest. The main difference is their convergence speed, which is higher with MAX-VIOLATION.

Because these strategies for global training operate on partial derivations, Aufrant et al. (2017) also propose a restart strategy, which augments both training strategies by pursuing training on the rest of the sentence, after each update. This provides the best convergence, and a slightly higher accuracy.

Oracle computation In another line of research, Goldberg and Nivre (2012) introduce dynamic oracles as another large improvement to local training. Instead of comparing with an arbitrary reference derivation (i.e. a static oracle), such oracles signal errors whenever an action prevents a gold dependency to be predicted later, in which case the action is said to have *non-zero cost*. Consequently, when training with dynamic oracles, erroneous actions are only judged at decoding time and this allows to accept as reference any derivation leading to the reference parse (non-determinism) and to compute reference actions even in suboptimal configurations (completeness). Deriving dynamic oracles is not necessarily straightforward and has only been done for a handful of transition systems (Goldberg and Nivre, 2013; Goldberg et al., 2014; Gómez-Rodríguez and Fernández-González, 2015).

Among others, dynamic oracles allow more evolved training strategies, like error exploration, which lets the parser learn to perform at best even when it has already strayed from the gold derivation.

Aufrant et al. (2017) extend dynamic oracles to beam search, by signaling errors when every hypothesis in the beam has accumulated one extra error since search started (from any arbitrary configuration), and then applying a global update. This methodology for oracle computation avoids explicit computation of the reference derivations, which is lighter and ensures soundness. See Appendix B for further details.

Enriched input Since they make the updates error-driven, dynamic oracles enable by design the use and processing of much more diverse datasets, with various kinds of sparsity or enrichment. Yet, straightforward implementations of action costs are in practice prone to relying on extra completeness assumptions, which allow easier cost descriptions but forbid sparsity. In PanParser, we have adopted an appropriate implementation which enables full use of the dynamic oracle properties.

First of all, dynamic oracles enable seamless training on examples with partial annotations, i.e. when only parts of the heads are specified: when no information is provided, the cost is simply under-estimated and no update occurs. This behavior holds even for more complex dynamic oracles, either global or non-arc-decomposable (see Appendix A).

Fine-grained subsampling can then be entertained by on-the-fly deletion of some reference dependencies, before training on a given example; error-driven training takes care of exploiting the remaining dependencies.

On another track, it is possible to exploit dynamic oracles to train and predict under constraints: restricting the search space to parses compatible with these constraints simply consists in restricting the legal actions to those that have zero cost with respect to the constraints. This way, the constraint dependencies are naturally respected and included by the parser, which in fact produces a standard derivation, without any pre- or post-processing.

As for partial parsers, which predict partial trees by design, thanks to the PanParser framework they are no different from standard parsers: they simply rely on modified transition systems, i.e. with relaxed preconditions and adapted action costs (see Appendix A), and training and prediction are done as usual.

4 Advanced uses

4.1 Local versus global implementations

We provide two implementations of the parser, respectively based on mutable and immutable objects.

The first implementation represents the state of the parser as a buffer pointer, a stack and a parse tree, that get updated whenever a transition is followed. This is a simple and efficient implementation for greedy parsing but its price is a loss of flexibility, and it can only be used to run fast baselines. In particular, because the parser state is mutable, using it with beam search would require many deep object copies and considerably slow down the process. Consequently, this implementation is only usable for greedy parsing with local training.

To make our parser usable with beam search, we also propose another implementation, that follows Goldberg et al. (2013): a parser state is just an immutable set of a few indexes and pointers to other parser states (previous state and tail of the stack).[2] Thus, derivations are represented as linked lists, and the complete information about a parser state (content of the stack, transition history, current parse tree) is distributed across all previous states, without duplicates. While accessing a deep stack element is necessarily slower than in the local implementation (and suboptimal for greedy parsing), factoring information in this way makes beam search and global training cheaper, both in time and memory usage.

The default implementation is global, it is controlled by the `beam` keyword, with 8 as a default beam size. Because of its higher flexibility and extra functionalities, we recommend this implementation.

```
# local implementation
parser = train_dependency_parser(dict(beam=None), dataset, n_epoch)
# equivalent to
parser = ArcEagerLocalParser(zn11_features)
train_dependency_parser(parser, dataset, n_epoch)

# global implementation
parser = train_dependency_parser(dict(beam=16), dataset, n_epoch)
# equivalent to
parser = ArcEagerBeamParser(zn11_features).with_beam(16)
train_dependency_parser(parser, dataset, n_epoch)
```

[2] Compared to their work, we enriched the representation of the stack and buffer pointers, with the current number of children, the two leftmost and the two rightmost children. This enables rich feature templates like Zhang and Nivre (2011)'s.

4.2 Transition system choice

PanParser includes several transition systems, which can be specified either by keyword or by class instance:

```
# ArcEager system
parser = train_dependency_parser(None, dataset, n_epoch)
# equivalent to
parser = ArcEagerBeamParser(zn11_features)
train_dependency_parser(parser, dataset, n_epoch)

# NonMonotonicArcEager system
parser = train_dependency_parser(dict(system="NonMonotonicArcEager"),
                                 dataset, n_epoch)
# equivalent to
parser = NonMonotonicArcEagerBeamParser(zn11_features)
train_dependency_parser(parser, dataset, n_epoch)

# ArcHybridSpan system
parser = train_dependency_parser(dict(system="ArcHybridSpan", easy_span=2),
                                 dataset, n_epoch)
# equivalent to
parser = ArcHybridSpanBeamParser(zn11_features).with_easy_span(2)
train_dependency_parser(parser, dataset, n_epoch)
```

Variants with the dummy ROOT token in first position are controlled by the `root_first` parameter:

```
parser = train_dependency_parser(dict(system="ArcEager", root_first=True),
                                 dataset, n_epoch)
parser = train_dependency_parser(dict(system="ArcHybrid", root_first=True),
                                 dataset, n_epoch)
parser = ArcHybridBeamParser(zn11_features).with_root_first(True)
train_dependency_parser(parser, dataset, n_epoch)
```

The user can also define its own transition system and use it with the standard API:

```
import custom_system
from dparser.perceptron_parser import PerceptronParser
class CustomBeamParser(custom_system.BeamParser, PerceptronParser): pass

# add to built-in scope
import dependency_parser
dependency_parser.CustomBeamParser = CustomBeamParser

parser = train_dependency_parser(dict(system="Custom", foo=bar), dataset, n_epoch)

# equivalent to
parser = CustomBeamParser(zn11_features).with_foo(bar)
train_dependency_parser(parser, dataset, n_epoch)

# or, if with_foo does not exist
parser = CustomBeamParser(zn11_features)
parser.foo = bar
train_dependency_parser(parser, dataset, n_epoch)
```

4.3 Oracle and training strategy

As they are often tightly interleaved, the training strategy and the oracle computation are both controlled by the same `strategy` parameter. PanParser has a few built-ins, based on static[3] and dynamic oracles, greedy and global training (with early update, max-violation, and full update). They also include restart options, and various exploration rates, i.e. the probability to pursue training from the erroneous configuration instead of the gold one.

```
# global training with the early update strategy
parser = train_dependency_parser(None, dataset, n_epoch, strategy="early")
```

[3] As most things in PanParser, static oracles are also computed using dynamic oracles. The reference derivation is built by pre-parsing the sentence while restricting the search space to zero cost actions. To ensure model independence, in this case beam hypotheses are not sorted by score but by action names.

The most useful built-ins are: `greedy` (static oracle), `goldberg` (Goldberg and Nivre, 2013), `greedyn` (dynamic oracle with full exploration), `early` and `maxv` (static oracles), `earlyNDrestartSO` and `maxvNDrestartSO` (which stand for 'non-deterministic restart sub-optimal', the improved strategies of Aufrant et al. (2017)) and `full` (global training with full update).

This diversity of strategies is allowed by the unified formalism adopted by PanParser (see Appendix B), in which the basic training unit is the model update, and all training strategies follow the same workflow: initialize a beam in a given configuration, extend the beam repeatedly until an error is flagged, select a pair of update configurations among the candidates, perform the update, iterate until the example is considered processed. This entertains greedy search by using a beam of size 1 and considering examples as processed only when a final configuration is encountered. Without restart the example is considered processed after the first update, with restart it requires final configurations. Errors can also be flagged for several reasons simultaneously, which allows non-standard criteria, like forcing the beam to reinitialize every few actions (thus preventing updates on very distant configurations).

The user is not restricted to the built-in strategies, and can also experiment with other combinations, as follows:

```
import dependency_parser
def strategies(parser, strategy, builtin=dependency_parser.strategies):
    (time_limit, check_cut_edge, check_single_reference, oracle, restart,
     exploration_rate, beam_update, remember_bad_beams) = builtin(parser, strategy)
    if strategy == "customstrategy":
        oracle = EARLY_LONGEST_PREFIX
        time_limit = 4
        restart = True
    return (time_limit, check_cut_edge, check_single_reference, oracle, restart,
            exploration_rate, beam_update, remember_bad_beams)
def exploration_strategy(strategy, itn, n_epoch,
                         builtin=dependency_parser.exploration_strategy):
    exploration_rate = builtin(strategy, itn, n_epoch)
    if strategy == "customstrategy":
        exploration_rate = 3*itn/epoch
    return exploration_rate
# replace built-in
dependency_parser.strategies = strategies
dependency_parser.exploration_strategy = exploration_strategy

parser = train_dependency_parser(None, dataset, n_epoch, strategy="customstrategy")
```

4.4 Partial training data

Training on partial annotations does not need to be activated, it is supported by default by all parsers using PanParser. However, it implies slight changes in data formats. In the Pan4 format, non-attached tokens are annotated with `None` instead of their head index. We similarly augment the CoNLL format by denoting empty attachments with '_', which the `read_conll` utility takes care of.

4.5 Constrained training and decoding

Dependency constraints also rely on data formats only, and PanParser detects their presence automatically. In a Pan4 example with constraints, the heads (or labels) list is replaced by a tuple, whose first element represents the gold standard and the second represents the constraints. At prediction time, only the constraints are read, and the first tuple element is ignored.

The `read_conll` function detects and builds such examples when the 9th column contains a constraint indication: '0' or '_' for 'the head is unknown', '1' for 'the head is known' (in which case the head specified in column 7 is added to the constraints). Note that this is not fully compatible with the CoNLL format, which sometimes uses column 9 for pseudo-projective parse indications.

4.6 Choice of classifier

Several built-in classifiers are provided: the averaged perceptron, a feedforward neural network, and generic classifiers for joint prediction and voting (which are in fact wrappers around other classifiers).

For instance, in order to have three parsers voting on each decision (with ties broken by parser1), first train the parsers separately, and then do:

```
from classifier.vote import VoteClassifier
parser1.model = VoteClassifier((parser1.model, 1.1), (parser2.model, 1),
                               (parser3.model, 1))
# now parser1 acts according to the vote of all three parsers
print(eval_dependency_parser((parser1, testset)))
```

The basic classifier API is based on the score and update functions. Other functions are used, typically predict, batch processing and after-training refinements (averaging), but they have vanilla implementations in the base class, so their implementation is optional to add a new classifier.

To create custom parser classes using a user-defined classifier, the AbstractParser class must be extended by implementing the _defaultmodel function. See dparser.perceptron_parser for a straightforward example.

4.7 Delexicalized and custom features

The feature function takes a configuration as argument and returns, in case of a linear classifier, a dictionary of string keys (the feature itself) with float values (mostly 1, for binary features). It only manages the feature templates: feature extraction is handled internally by the parser and is not customizable yet.

If its embeddings parameter is set (to a dictionary mapping strings to real-valued iterables), the embeddings of some words (top two stack elements, buffer head) are appended to the feature vector, using arbitrary keys for each (real-valued) dimension. The embedding dictionary must at least contain the ‘<UNK>’ key, which denotes the default embedding of unknown words.

The zn11_features function also has a morpho option (default: false) which extends the feature representation with morphological tags from the stack and buffer heads.

Additionally, the delexicalized parameter controls whether the lexicalized parameters are used and updated. This is particularly useful for cross-lingual parsing. The value can notably be changed on an already trained parser, which can be useful for fine-tuning.

```
from functools import partial

# extend binary features with embeddings
embeddings = {"<UNK>": [0,0.9,0.5,1], "DeLorean": [1,0,0.1,0],
              "Chicken": [0.8,0.1,0.3,1]}
custom_features = partial(zn11_features, embeddings=embeddings)
parser = ArcEagerBeamParser(custom_features)

# use delexicalized features with morphological tags
custom_features = partial(zn11_features, delexicalized=True, morpho=True)
parser = ArcEagerBeamParser(custom_features)
train_dependency_parser(parser, dataset, n_epoch)
# fine-tuning with lexicalized parameters
parser.features = partial(parser.features, delexicalized=False)
train_dependency_parser(parser, other_dataset, 1)
```

Note that the embeddings and delexicalized parameters can also be specified in the parser initialization parameters:

```
parser = train_dependency_parser(dict(delexicalized=True), dataset, n_epoch)
```

Using the parser creation by class instance, any user-defined feature function can of course be used instead of zn11_features.

4.8 Labeled parsing

Labeled parsing is also possible with PanParser. The labeled parsers typically extend the LabeledPerceptronParser class, which internally uses the built-in JointClassifier: labels are predicted in parallel to action predictions, using the same feature representations.

The use of the labeling component can be specified as part of the transition system name:

```
parser = train_dependency_parser(dict(system="ArcEagerLabeled"), dataset, n_epoch)
# equivalent to
parser = ArcEagerLabeledBeamParser(zn11_features)
train_dependency_parser(parser, dataset, n_epoch)
```

Labeled parsers process Pan4 data similarly to unlabeled parsers, but additionally fill the list of labels, instead of defaulting to `None`.

4.9 Tagger

PanParser has a built-in PoS tagger, based on the same structured prediction framework. It shows how this framework can be used for other structured prediction tasks than dependency parsing. This unification also paves the way to joint tagging and parsing with PanParser.

The structure and usage of the tagger are similar to PanParser, albeit simpler because it does not involve transition systems. The main difference is that at training time, the tagger also builds a tag dictionary of unambiguous words, with almost always the same tag (and enough occurrences) in the dataset, and at prediction time it tries looking up the tag in the dictionary, before defaulting to actual predictions.

The tagger does not use Pan4 but the Pan2 format, i.e. only words and tags, without ROOT padding. It is still possible to use it with Pan4 data, which is automatically detected and internally reformated, but tag and feature dictionaries must first be converted to strings, with the `feat2tag` function retaining coarse tags by default.

```
from dependency_parser import read_conll
with open("en-train.conllu", "rt") as fh:
    dataset = read_conll(fh)
with open("en-test.conllu", "rt") as fh:
    testset = read_conll(fh)

# train and evaluate a perceptron tagger
from tagger.postagger import train_pos_tagger, eval_pos_tagger
from tagger.perceptron_postagger import PerceptronPOSTagger
from conll.io import feat2tag
tagger = train_pos_tagger(PerceptronPOSTagger(), feat2tag(dataset), n_epoch)
print(eval_pos_tagger((tagger, feat2tag(testset))))

# tagging and parsing pipeline
from tagger.neural_postagger import FeedForwardPOSTagger
tagger = train_pos_tagger(FeedForwardPOSTagger(), feat2tag(dataset), n_epoch_tag)
# to train on predicted tags
from tagger.postagger import process_pos_tagger
dataset = process_pos_tagger(tagger, dataset)
parser = train_dependency_parser(None, dataset, n_epoch_parse)
# annotate new data
with open("en.conllu", "rt") as fh:
    newdataset = read_conll(fh)
from conll.io import wt, tag2col
with open("out.conllu", "wt") as out:
    wt(out, process_pos_tagger(tagger, newdataset), preprocess=tag2col)
```

4.10 Error analysis

The `eval_dependency_parser` and `eval_pos_tagger` functions are built on evaluation utilities provided by the core framework. On top of overall accuracies, they can also provide fine-grained statistics and scores.

Several fine-grained criteria are built in, but they can be further combined. The following shows a series of examples of what these functions allow in error analysis.

```
# ignore PUNCT (default), INTJ, SYM and X tokens in evaluation
eval_dependency_parser((parser, testset), ignore_tags=["PUNCT", "INTJ", "SYM", "X"])

# fine-grained scores depending on the (reference) in-tree depth of the tokens,
# capped to 4
eval_dependency_parser((parser, testset), by_type=MDEPTH)
```

```
# fine-grained scores for PoS tag pairs, eg. attachment score over the nouns whose
# head is a verb
from misc.xp import uas_bitable
uas_bitable(eval_dependency_parser((parser, testset), by_type=(POS, HEADPOS)))
# the same, but only for tokens whose head is on the right
uas_bitable(eval_dependency_parser((parser, testset), by_type=(POS, HEADPOS),
                                   subset=(DIRECTION, "+")))

# fine-grained scores for all PoS-disambiguated words with "mark" attachment, i.e.
# keys are "except-ADP", "except-SCONJ", "like-ADP", "like-SCONJ", "with-ADP", etc.
from misc.xp import scores
scores(eval_dependency_parser((parser, testset), by_type=[WORD, "-", POS],
                              subset=(LABEL, "mark")))

# compute corpus statistics both on the direction-disambiguated head PoS for each
# child PoS ("2pos+dir" key), and on the dependency length ("L" key)
eval_dependency_parser(testset, by_type={"L": LENGTH,
                       "2pos+dir": (POS, [HEADPOS, DIRECTION])}, out=STATS)

# confusion matrix on the head PoS
from misc.xp import confusion_matrix
confusion_matrix(eval_dependency_parser((parser, testset), by_type=HEADPOS,
                                        out=CONFUSION))

# attachment agreement of two parsers, for each gold label
eval_dependency_parser([(p1, testset), (p2, testset)], by_type=LABEL, out=COMPARE)[2]
```

5 Code organization

5.1 Structured prediction framework

The whole framework revolves around the `AbstractStructuredPredictor` class, from which all taggers and parsers inherit.

Its design is based on the idea of coroutines: the components interact via Python generators, each representing a different kind of data (prediction configurations, update configurations, predicted classes, output annotations, etc., as well as batched versions of all). This is what allows the classifier to be considered as a black box: it simply feeds from a stream of configurations, and provides a stream of predictions. The role of the `AbstractStructuredPredictor` class is to plug these generators together, and to add a feedback loop so that a given prediction can be both used as an output, and reused (by the task-specific component) to compute the next configuration.

Task-specific components Designing a task-specific component consists in implementing two methods. The `xs` method returns (given an input sentence) a function mapping a stream of classifier predictions to a (multiplexed) stream of feature vectors to classify and of output predictions (eg. a tag or the k-best parse trees). `xys` does the same for training, i.e. yielding feature vectors to classify, as well as update pairs. Additionally, `polish` can be overridden: it does nothing by default, but can add post-processing of the final output, for instance gathering all tags into a list or selecting the top hypothesis of a k-best list.

The base class has another method to implement, `_defaultmodel`, but this is done when plugging the wanted classifier: it builds and returns a new classifier, initialized with the dataset.

Framework API A system built with this framework can then be used with only two methods, `train` and `process`. The former takes care of epochs, sampling, cross-validation and post-training refinements like averaging. The latter initiates dataset annotation, with pre- and post-processing to fill the appropriate slots with the tags, heads or labels predictions.

System customization There are two ways to add optional behaviors or parameters. The first is the `strategy`, `epoch_strategy` and `sample_strategy` methods: based on a strategy identifier and knowledge of the epoch, they return additional training parameters that

AbstractStructuredPredictor ultimately feeds to xys. By default they are placeholders, overwritten by custom functions in train_dependency_parser and train_pos_tagger.

The second is a series of hooks, gathered in a Callback object sent to the train, xys and xs methods. They are called in various crucial places (before and after epochs, sample processing, updates, evaluation, etc.) and can be used for additional monitoring, or for more evolved purposes like on-the-fly subsampling.

Evaluation utilities The AbstractStructuredPredictor class also provides several basic utilities for evaluation and error analysis. The eval_routine function collects fine-grained accuracies for arbitrary category criteria, and matrix_routine builds a confusion matrix with such arbitrary groupings. compare_routine and matrix_compare_routine compare these measures between two systems. Finally, stats_routine provides various statistics on the dataset itself, using the same kind of criteria. All these functions operate on annotated datasets; testset annotation is rather handled at the level of eval_dependency_parser and eval_pos_tagger.

5.2 Beam search and global training strategies

Beam parsers are also based on the structured prediction framework, but they require extra generic functionalities, which the AbstractBeamStructuredPredictor class provides.

The lookahead method is in charge of the actual beam search. Given a stream of classifier predictions and an initial state for the beam (in most cases, a single configuration), it repeatedly extends the beam until an erroneous state is found, then yields all subsequent states, until final configurations. It consequently returns a multiplexed generator of feature vectors to classify, and erroneous states (i.e. k-best hypotheses). Using generators makes decoding lazy, so that in case of early update it can stop after the first error and no extra computation is done. At each extension of the beam, lookahead also enriches the hypotheses with aggregated information on the action costs, so that erroneous states are detected in constant time. Note that the gold actions are those with minimum cost and not zero cost: Appendix A explains why.

The _infer method is in charge of searching for update configuration pairs. Given a stream of classifier predictions and a sentence, it repeatedly calls lookahead (initially, on an empty configuration), processes the erroneous states to find an update pair, then selects a new initial configuration and goes on, until the sentence is processed. Here again, restart is lazy. To select update configurations, _infer applies the specified training strategy, thanks to three oracle utilities: forced_decoding to perform a lookahead with gold actions only, lookfurther to search for a max-violation state among those returned by lookahead, and refchoice to select a positive configuration when there are several.

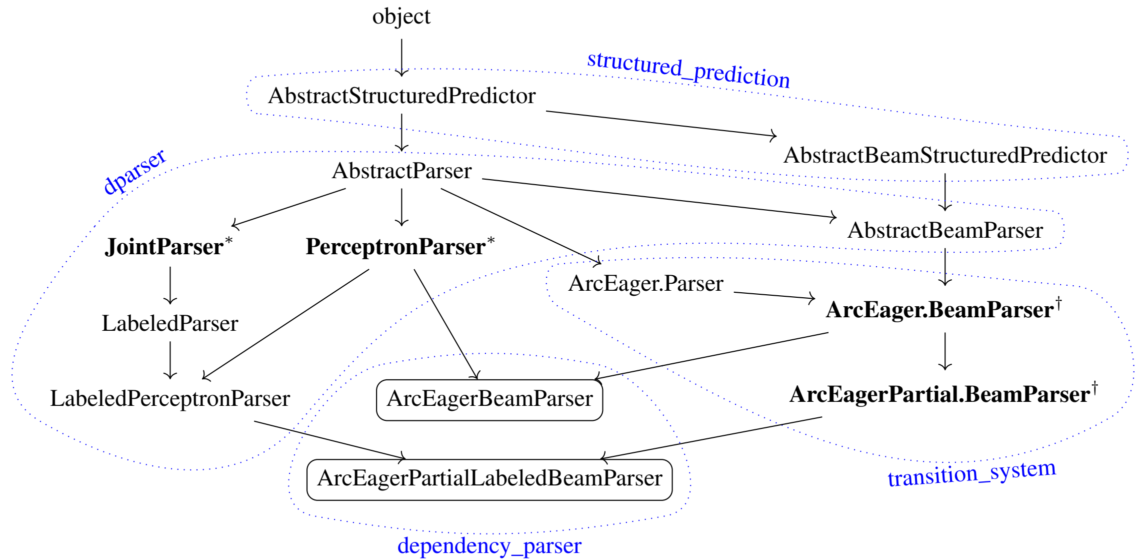
When no error criterion is given (at prediction time), by design lookahead and _infer perform a single decoding pass, until full processing.

In the AbstractBeamParser class, the search and learn methods (returned respectively by xs and xys) both use _infer. search gets a beam of final hypotheses and extracts the k-best parse trees from these derivations. learn, for each update pair, goes through their history to extract feature vectors for classifier updates.

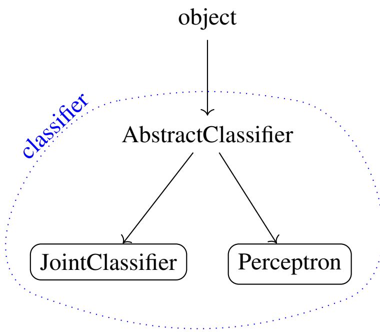
5.3 Class hierarchy

Because it focuses on modularity and maximal code factorization, PanParser contains a lot of classes, each bringing an additional piece of parser design. For instance, the ArcEager.Parser class registers the set of transitions into the parser, forwarded to the classifier as possible classes by PerceptronParser, ArcEager.TransitionSystem defines the legal transitions in a given configuration and their action cost, while ArcEager.BeamParse provides their semantics, i.e. the effect of each action on the parse configuration, and how to output a parse tree based on the derivation.

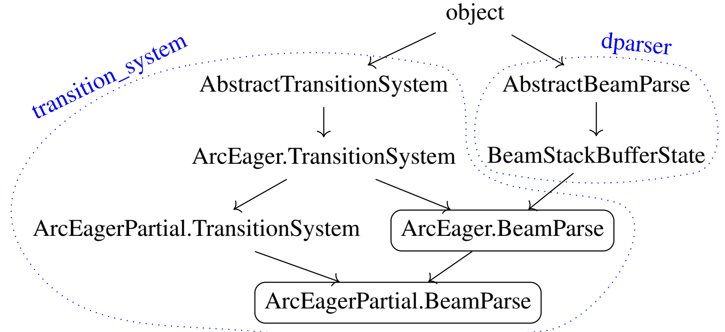
For this reason, and as a reference, we picture in Figure 2 the full class hierarchy of two parser classes: ArcEagerBeamParser for a standard case, and ArcEagerPartialLabeledBeamParser for a particularly complex case.



(a) Parser classes.



(b) Classifier classes.



(c) Parse configuration classes.

Figure 2: Class hierarchy for the `ArcEagerPartialLabeledBeamParser` and `ArcEagerBeamParser` classes, which are those actually used to build and train parsers. Classes denoted with * contain references to classifier classes (`JointClassifier` and `Perceptron`). Classes denoted with † contain references to parse configuration classes (`BeamParse` classes). Package names are indicated in blue.

6 Conclusion and future work

We have presented PanParser, a transition-based dependency parser implemented with the concern of algorithmic variation completeness, intended both for practical uses and as a parsing research tool. It currently supports numerous options and customizations for several aspects of the parsing algorithms.

PanParser is, however, still a work in progress, and we already plan several extra developments. We intend to take a further step to customization completeness, by allowing to parse without dummy ROOT token, and to extract features from both the constraints and the predicted labels. We will also add built-in transition systems that are not stack- and buffer-based: the COVINGTON system, based on two lists and a buffer, and for which Gómez-Rodríguez and Fernández-González (2015) already derived a dynamic oracle; the SWAPSTANDARD system (using a stack and a list), which requires deriving new efficient dynamic oracles; and the EASYFIRST system, based on a single list. Another planned extension is to add relaxed types of arc constraints, eg. ambiguous constraints, and span constraints.

Finally, we will add support for stateful classifiers to add a stack-LSTM parser implementation, and allow arbitrary joint prediction, which should achieve full PanParser support for state-of-the-art systems like that of Swayamdipta et al. (2016).

References

Lauriane Aufrant, Guillaume Wisniewski, and François Yvon. 2016. Ne nous arrêtons pas en si bon chemin: améliorations de l'apprentissage global d'analyseurs en dépendances par transition. In *Actes de la 23e conférence sur le Traitement Automatique des Langues Naturelles*, pages 248–261.

Lauriane Aufrant, Guillaume Wisniewski, and François Yvon. 2017. Don't Stop Me Now! Using Global Dynamic Oracles to Correct Training Biases of Transition-Based Dependency Parsers. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 318–323.

Miguel Ballesteros and Joakim Nivre. 2013. Going to the Roots of Dependency Parsing. *Computational Linguistics*, pages 5–13.

Michael Collins and Brian Roark. 2004. Incremental Parsing with the Perceptron Algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 111–118.

Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*, pages 959–976.

Yoav Goldberg and Joakim Nivre. 2013. Training Deterministic Parsers with Non-Deterministic Oracles. *Transactions of the Association for Computational Linguistics*, pages 403–414.

Yoav Goldberg, Kai Zhao, and Liang Huang. 2013. Efficient Implementation of Beam-Search Incremental Parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 628–633.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. 2014. A Tabular Method for Dynamic Oracles in Transition-Based Parsing. *Transactions of the Association for Computational Linguistics*, pages 119–130.

Carlos Gómez-Rodríguez and Daniel Fernández-González. 2015. An Efficient Dynamic Oracle for Unrestricted Non-Projective Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 256–261.

Matthew Honnibal, Yoav Goldberg, and Mark Johnson. 2013. A Non-Monotonic Arc-Eager Transition System for Dependency Parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured Perceptron with Inexact Search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic Programming Algorithms for Transition-Based Dependency Parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-Projective Dependency Parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 99–106.

Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*.

Joakim Nivre. 2004. Incrementality in Deterministic Dependency Parsing. In *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57.

Swabha Swayamdipta, Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2016. Greedy, Joint Syntactic-Semantic Parsing with Stack LSTMs. *arXiv preprint arXiv:1606.08954*.

Yue Zhang and Stephen Clark. 2008. A Tale of Two Parsers: Investigating and Combining Graph-based and Transition-based Dependency Parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571.

Yue Zhang and Joakim Nivre. 2011. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193.

Yue Zhang and Joakim Nivre. 2012. Analyzing the Effect of Global Learning and Beam-Search on Transition-Based Dependency Parsing. In *Proceedings of COLING 2012: Posters*, pages 1391–1400.

A Transition systems and dynamic oracles

A.1 Defining a transition system

Adding a new transition system is straightforward and can be done in a custom separate file. See the ARCEAGER description for a straightforward example. The system definition must extend one of the abstract parser classes by specifying: (a) the valid actions in any configuration, (b) the effect of each action, (c) the dynamic cost of each action, and (d) for the beam implementation only, how to iterate on the ancestors of the first element in stack, and how to retrieve the parse tree based on transition history.

Defining the dynamic cost is sufficient for using the transition system with any training option. However, for some transition systems, the cost of an action may be difficult to express, or computationally too expensive. In this case, it is still possible to define the cost as a degenerated version of a static oracle: the transition heuristically designated in the pre-computed reference is given cost 0, all other transitions are given cost 1. This method ensures that any transition system can be incorporated in the PanParser, but keep in mind that in this case it will not be fully compatible with other components (no dynamic oracle, so no error exploration and no constrained parsing).

A.2 Action cost computation

To specify dynamic oracles, Goldberg and Nivre (2012) formally define the cost of an action as 'the loss difference between the minimum loss tree reachable before and after' performing the action in question. Considering the minimum loss is equivalent to the maximum UAS achieved on the given example. So, the cost is evaluated by computing the maximum UAS over all derivations resulting from the given configuration (c), and the maximum over only those following the given action (t).

$$\textsc{Cost}(c,t) = \left[\max_{t_1,\cdots,t_{final}} \text{UAS}(c \circ t_1 \circ \cdots \circ t_{final}) \right] - \left[\max_{t_2,\cdots,t_{final}} \text{UAS}(c \circ t \circ t_2 \circ \cdots \circ t_{final}) \right]$$

By definition of the maximum, COST is always positive, and in every configuration at least one action has zero cost.

Arc-decomposable systems Exhaustively exploring all the successor derivations is computationally too expensive, and thus Goldberg and Nivre (2013) define the notion of arc-decomposition to simplify cost computation. A transition system is *arc-decomposable* if for any configuration c, all arcs reachable from c (i.e. predicted by at least one transition sequence after c) can be reached conjointly by a single transition sequence. This means that at the level of the transition system, there is no interaction between predicted arcs, and no incompatibility effect.

If we define FORBIDDENARCS(c,t) as the number of arcs that are reachable from c but not from $c \circ t$, Goldberg and Nivre (2013) state that for arc-decomposable systems, these arcs are the only source of cost, and thus:

$$\textsc{Cost}(c,t) = \textsc{ForbiddenArcs}(c,t)$$

Non-arc-decomposable systems When this property does not hold, on the other hand, there are extra sources of cost to account for, because of incompatible arcs. In case of such incompatibilities, at some point, adding a gold arc will indeed imply renouncing to another gold arc, thereby inserting an error. But this cost cannot be attributed to the given action, it is in fact due to a much earlier action, which introduced the incompatibility. Besides, sometimes in such cases, FORBIDDENARCS is non-zero for all legal actions, in which case it is obviously not identical to the COST function.

There are two main strategies to compute the action cost in non-arc-decomposable systems. The first is to explicitly compute the loss before and after the action, typically using dynamic programming (Goldberg et al., 2014), and then retain the difference. The second is to directly model the cost, by formalizing the configurations holding arc incompatibilities, and detecting when such incompatibilities are inserted (Gómez-Rodríguez and Fernández-González, 2015). When possible, this is computationally cheaper than a full loss computation.

Relaxed action cost To formalize the cost in a non-arc-decomposable system, we define $\textsc{ExpectedCost}(c)$ as the number of arcs that are still reachable from c but do not belong to the final output (considering some final configuration, reachable from c and with maximal UAS). This counts the number of current incompatibilities. The action cost then decomposes as:

$$\textsc{Cost}(c,t) = \textsc{ForbiddenArcs}(c,t) + (\textsc{ExpectedCost}(c \circ t) - \textsc{ExpectedCost}(c))$$

We now introduce the $\textsc{RelaxedCost}$ function, defined as:

$$\textsc{RelaxedCost}(c,t) = \textsc{ForbiddenArcs}(c,t) + \textsc{ExpectedCost}(c \circ t)$$

from which ensues:

$$\begin{aligned} \textsc{Cost}(c,t) &= \textsc{RelaxedCost}(c,t) - \textsc{ExpectedCost}(c) \\ \textsc{ExpectedCost}(c) &= \textsc{RelaxedCost}(c,t) - \textsc{Cost}(c,t) \leq \textsc{RelaxedCost}(c,t) \end{aligned}$$

and because at least one action has zero cost:

$$\begin{aligned} \textsc{ExpectedCost}(c) &= \min_t \textsc{RelaxedCost}(c,t) \\ \textsc{RelaxedCost}(c,t) &= \textsc{ForbiddenArcs}(c,t) + \min_{t'} \textsc{RelaxedCost}(c \circ t,t') \\ \textsc{Cost}(c,t) &= \textsc{RelaxedCost}(c,t) - \min_{t'} \textsc{RelaxedCost}(c,t') \end{aligned}$$

In other words, the $\textsc{RelaxedCost}$ function computes an overestimate of $\textsc{Cost}$, that repeatedly counts the cost of incompatibilities, as long as they are not resolved, and not only when they are introduced. Thus, it may happen that no action has a zero $\textsc{RelaxedCost}$, but the actual cost can be retrieved by shifting all costs by the minimum $\textsc{RelaxedCost}$, which corresponds to the current $\textsc{ExpectedCost}$. Hence, in this framework, the optimal actions are not those with zero cost but with minimal cost.

These definitions have notably two useful properties, which make the use of the alternate definition transparent. First, for arc-decomposable systems, $\textsc{ExpectedCost}$ is null, so $\textsc{RelaxedCost} = \textsc{Cost}$. Second, since $\min_t \textsc{Cost}(c,t) = 0$, in both cases ($\textsc{RelaxedCost}$ and $\textsc{Cost}$), shifting by the minimum cost always yields $\textsc{Cost}$ values.

Consequently, from now on, we define optimal actions as *minimum-cost actions*, and transition system implementations are supposed to compute either one of $\textsc{Cost}$ (when computed as a loss difference) and $\textsc{RelaxedCost}$ (when modeled explicitly).

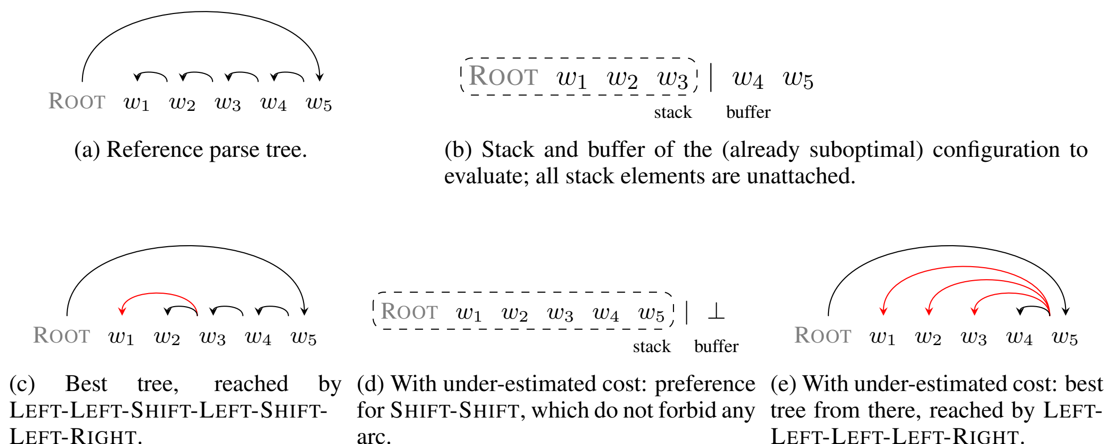
In practice, defining the action cost explicitly then consists in listing as usual the arcs that the action makes unreachable, as well as the causes of arc incompatibilities in the future configuration.

Consequences of under-estimated costs Exhibiting all causes of incompatibilities is often a tedious task, it is even more so to *prove* that the list is exhaustive, as done by Gómez-Rodríguez and Fernández-González (2015) for the Covington system. We have not yet done this study for all non-arc-decomposable systems in PanParser, and have settled for now for firm beliefs: the non-arc-decomposable costs have indeed been tested against exhaustive search on all possible configurations, but for short sentences only.

So, what happens if we have missed a given type of incompatibility? Or worse, if we miss all of them and simply use $\textsc{ForbiddenArcs}$ for a non-arc-decomposable system? Using minimum-cost instead of zero-cost actions in fact strongly alleviates such issues.

Indeed, with an under-estimated cost, some actions introducing incompatibilities may be deemed correct, later resulting in configurations where all actions forbid some reference arc, even though no error has been detected in the past. With standard cost definition and a zero-cost criterion, this case is not covered, and training would presumably stop. But with the minimum-cost criterion there are always gold actions, whether the cost is correctly defined or not, and training can consequently go on transparently.

The only consequence on training is that the cost under-estimation introduces for the oracle a preference towards late resolution of inconsistencies: in case of two incompatible arcs, the parser will prefer



(a) Reference parse tree.

(b) Stack and buffer of the (already suboptimal) configuration to evaluate; all stack elements are unattached.

(c) Best tree, reached by LEFT-LEFT-SHIFT-LEFT-SHIFT-LEFT-RIGHT.

(d) With under-estimated cost: preference for SHIFT-SHIFT, which do not forbid any arc.

(e) With under-estimated cost: best tree from there, reached by LEFT-LEFT-LEFT-LEFT-RIGHT.

Figure 3: Consequences of training with an under-estimated cost ignoring arc incompatibilities, using ARCSTANDARD with ROOT in first position.

actions that keep both options as long as possible over actions that forbid one of them right away. Figure 3 shows how bad this tendency can be. However, whether such cases have a strong impact on accuracy remains to be ascertained.

Consequently, the minimum-cost criterion makes it possible to use under-estimated costs, typically by ignoring non-arc-decomposability, but such unsound training has unknown consequences on accuracy.

A.3 Non-projectivity

Because dynamic oracles have the ability to *accept* past errors and do their best to select good decisions anyway, by design they make it possible to train on non-projective sentences even with a projective transition system. The issue of non-projectivity is indeed exactly the same as that of arc incompatibilities: when two crossing edges are reachable, only one can actually belong to the final output.

Hence, from the oracle point of view, the initial empty configuration already comes with embedded 'past errors' (the incompatibilities due to edge crossings). As for non-arc-decomposable systems, the cost incurred by these incompatibilities is not due to actions to come, but should be attributed to previous actions, taken in a fictive history before the initial configuration. As such, the natural behavior of dynamic oracles is to ignore this cost.

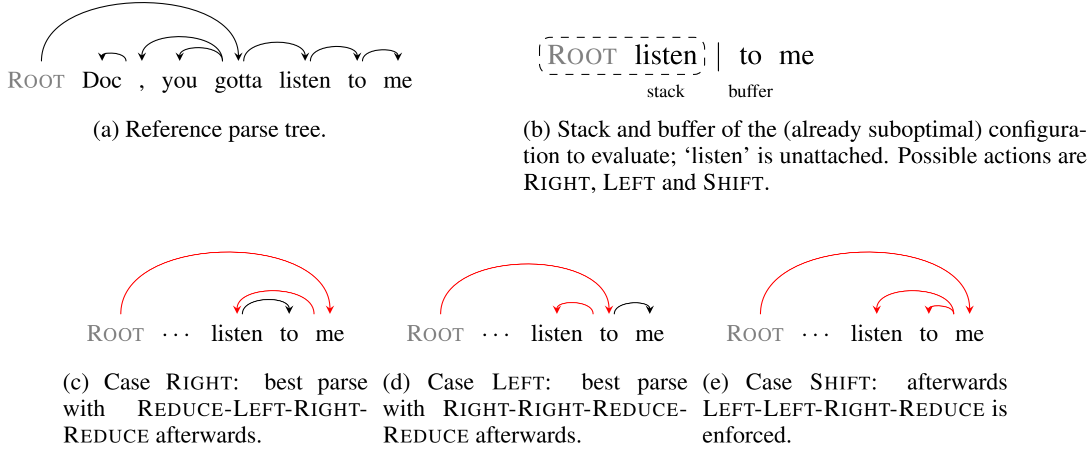
The costs of built-in transition systems have not been adapted yet to acknowledge arc incompatibilities due to non-projectivity. For now, we consequently use under-estimated costs for those sentences, which empirically remains better than discarding all non-projective sentences.[4] An interesting track for future work would be to compare empirically this method for exploiting non-projective sentences with methods based on pseudo-projectivization (Nivre and Nilsson, 2005).

A.4 ROOT position

A specific concern must be addressed to the place of the dummy ROOT token. By default in last position in PanParser, it can be put in front by calling the *with_root_first* method of the parser before training starts. Ballesteros and Nivre (2013) show however that it is not a mere technical choice but leads to varying accuracies depending on the position of this token and the transition system. The difference goes in fact beyond that and it interferes with valid actions, since some of them can involve dummy tokens, and consequently with cost computation for dynamic oracles.

To illustrate this, consider the ARCEAGER system (with ROOT in final position), proved arc-decomposable by Goldberg and Nivre (2013), i.e. in any configuration, all the reference arcs that are still reachable can be predicted together. However, when placing the ROOT token in front of the sentence, the proof does not hold anymore: Figure 4 presents a counter-example to arc-decomposability.

[4] In UD 2.0, we have measured gains of +5 UAS for the poorly projective Ancient Greek.



(a) Reference parse tree.

(b) Stack and buffer of the (already suboptimal) configuration to evaluate; 'listen' is unattached. Possible actions are RIGHT, LEFT and SHIFT.

(c) Case RIGHT: best parse with REDUCE-LEFT-RIGHT-REDUCE afterwards.

(d) Case LEFT: best parse with RIGHT-RIGHT-REDUCE-REDUCE afterwards.

(e) Case SHIFT: afterwards LEFT-LEFT-RIGHT-REDUCE is enforced.

Figure 4: Counter-example to arc-decomposability of the ARCEAGER transition system with ROOT in first position.

A.5 Built-in transition systems

In the following, we document all transition systems built in PanParser, along with their cost. s and s' denote top elements of the stack σ, b denotes the head of buffer β, and P is the set of edges representing the partially built tree. The tokens that are not words are the padding tokens. For the action costs, we also provide compact representations of each setting yielding non-zero costs, in which case σ represents any element in σ (same for β) and h^*_w is the reference head of word w.

The partial and short-spanned systems are new transition systems designed as part of PanParser. All others are drawn from the literature, but in general their properties had not yet been studied for both ROOT positions.

A.5.1 ArcEager

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P+(b\rightarrow s))$	if s is a word and s is unattached
RIGHT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s\|b,$	$\beta,$	$P+(s\rightarrow b))$	
REDUCE	$(\sigma\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$\beta,$	$P)$	if s is attached

Table 2: ARCEAGER transition system.

SHIFT	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT	$(\sigma\|s, b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT	$(\sigma,\ b\|\beta)$	$b \curvearrowleft \beta$	b if h^*_b is in buffer but not on top
	$(\sigma\|s, b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
REDUCE	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 3: ARCEAGER dynamic cost: tokens that explicitly lose their head.

Case with ROOT in first position When the ROOT token is in first position, the upper description of the ARCEAGER system (Nivre, 2004) requires some amendments.[5]

First, if affects the preconditions of some actions. Indeed, SHIFT becomes illegal when the buffer contains a single element, because afterwards it cannot be attached, and the final configuration with ROOT token requires an empty stack. Similarly, RIGHT is illegal when the buffer contains a single element and at least one word in the stack is unattached.

Second, as shown in §A.4, it makes the system non-arc-decomposable. There is exactly one configuration which embeds arc incompatibilities. It is the case when the reference ascendance of the last word in sentence (n_N) consists in any number of buffer tokens followed by a stack element, and deeper in the stack (including n_N's ancestor) at least one word is still unattached. Because of the extra preconditions, in that case at least one ancestor of n_N will not get its correct head, but will instead be promoted as head of the unattached stack element. Figure 5 illustrates this configuration.

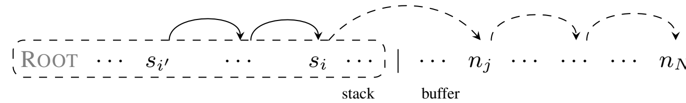


Figure 5: Prototype of arc incompatibilities for ARCEAGER with ROOT in first position.

Consequently, non-arc-decomposability adds a (relaxed) cost of 1 if the parser is in this configuration, or will be put in this configuration by the given action. More precisely, a single arc incompatibility is inserted the first time that an ancestor of the last token is shifted, or attached to a stack containing at least one unattached word.

Additional cost: head direction constraints As an experimental feature, the ARCEAGER system provides the possibility to compute the action cost when only the dependency direction is known. This mostly allows to parse under simpler constraints, for instance built using typological knowledge. Table 4 documents the additional cost incurred by such annotations.

SHIFT	$(\sigma\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h_b^* is on the left and the stack is not empty
LEFT	$(\bot\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h_b^* is on the left and the stack contains a single token
RIGHT	$(\sigma,\ b\|b'\|\beta)$	$b \curvearrowleft ?$	b if h_b^* is on the right and the buffer contains at least two tokens
REDUCE	$(\bot\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h_b^* is on the left and the stack contains a single token

Table 4: ARCEAGER dynamic cost: tokens whose specified head direction is explicitly forbidden.

A.5.2 ArcHybrid

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \to s))$	if s is a word
RIGHT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \to s))$	

Table 5: ARCHYBRID transition system.

The ARCHYBRID system has been proposed by Kuhlmann et al. (2011) as a compromise between ARCSTANDARD and ARCEAGER properties.

In this system, changing the ROOT position has no effect on preconditions, since the top of the stack can always be attached, either on the left or on the right. The cost is also unchanged, as the ARCHYBRID system is always arc-decomposable.

[5]To the best of our knowledge, this topic is not addressed in the literature.

SHIFT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\quad b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\quad \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT	$(\sigma\|s,\quad \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s,\quad \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 6: ARCHYBRID dynamic cost: tokens that explicitly lose their head.

A.5.3 ArcStandard

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	
LEFT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s,$	$\beta,$	$P + (s \rightarrow s'))$	if s' is a word
RIGHT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \rightarrow s))$	

Table 7: ARCSTANDARD transition system.

For the ARCSTANDARD system, originally designed by Nivre (2003), note that there is no precondition on SHIFT. Indeed, when the ROOT token is in last position, it must be shifted to receive its children and reach a final configuration. The dynamic oracle for this system directly computes COST as a loss difference, following the methodology proposed by Goldberg et al. (2014).

A.5.4 NonMonotonicArcEager

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \rightarrow s))$	if s is a word
RIGHT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s\|b,$	$\beta,$	$P + Temporary(s \rightarrow b))$	if b is a word
REDUCE	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \rightarrow s))$	

Table 8: NONMONOTONICARCEAGER transition system.

SHIFT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\quad b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\quad \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\quad b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
REDUCE	$(\sigma\|s,\quad \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s,\quad \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 9: NONMONOTONICARCEAGER dynamic cost: tokens that explicitly lose their head.

The NONMONOTONICARCEAGER system (Honnibal et al., 2013) is in fact very similar to ARCHYBRID, with the RIGHT transition renamed as REDUCE and a second shift transition, called RIGHT. However, even if both have the same expressivity and search space, at feature level the NONMONOTONICARCEAGER system allows to enrich the representation with knowledge of highly probable heads, which may help some decisions.

Still, this does not affect the action cost, which is the same as for ARCHYBRID. Table 10 provides additional costs for head direction constraints, similarly to the ARCEAGER case.

LEFT	$(\bot\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h_b^* is on the left and the stack contains a single token
	$(\sigma\|s'\|s,\ \beta)$	$? \curvearrowright s$	s if h_s^* is on the left and the stack contains at least two tokens
REDUCE	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft ?$	s if h_s^* is on the right and the buffer is not empty
	$(\bot\|s,\ b\|\beta)$	$? \curvearrowright b$	b if h_b^* is on the left and the stack contains a single token

Table 10: NONMONOTONICARCEAGER dynamic cost: tokens whose specified head direction is explicitly forbidden.

A.5.5 ArcEagerPartial

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \to s))$	if s is a word and s is unattached
RIGHT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s\|b,$	$\beta,$	$P + (s \to b))$	
REDUCE	$(\sigma\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$\beta,$	$P)$	if s is a word

Table 11: ARCEAGERPARTIAL transition system.

SHIFT	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h_b^* is in stack
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT		✗ $\curvearrowright s \curvearrowleft$ ✗	s if no h_s^*
	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h_s^* is in buffer but not on top
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT		✗ $\curvearrowright b \curvearrowleft$ ✗	b if no h_b^*
	$(\sigma,\ b\|\beta)$	$b \curvearrowleft \beta$	b if h_b^* is in buffer but not on top
	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h_b^* is in stack but not on top
	$(\sigma,\ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
REDUCE	$(\sigma\|s,\ \beta)$	$s \curvearrowleft \beta$	s if h_s^* is in buffer and s is unattached
	$(\sigma\|s,\ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 12: ARCEAGERPARTIAL dynamic cost: tokens that explicitly lose their head.

ARCEAGERPARTIAL is our proposal for an ARCEAGER parser that learns to *reproduce* partial annotations, i.e. it both learns syntactic knowledge based on the provided annotations, and learns which tokens should remain unannotated because of a lack of information. This is close to confidence-based learning, except that the confidence criterion is embedded in the training data.

In practice, the ARCEAGERPARTIAL system extends ARCEAGER with the possibility to predict empty attachments as part of the system, and the final output is consequently a partial tree by design.

Empty attachments are encoded by SHIFT+REDUCE: the word is put on the stack as usual, can receive children, but then is popped before receiving its own head. Consequently, the only modification to the system semantics is to relax the precondition on REDUCE actions, and allow them even on unattached words.

The impact is stronger on the action cost. Compared to ARCEAGER, it indeed adds two cases of non-zero cost: when a word whose reference attachment is empty receives a head (to enforce reproduction of empty attachments), and when an unattached word is reduced while its reference head is still somewhere in the buffer (which cannot happen with ARCEAGER).

Note that when the dynamic oracle is used to filter actions based on partial constraints, the cost incurred by empty attachments is ignored, otherwise the partial tree could never be completed.

Finally, regarding the ROOT position, it does not affect ARCEAGERPARTIAL like it does for ARCEAGER, because with the relaxed precondition, it is always possible to reduce stack elements to reach a final configuration. The system is consequently always arc-decomposable.

A.5.6 ArcHybridPartial

SHIFT	$(\sigma,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma\|b,$	$\beta,$	$P)$	if b is a word
LEFT	$(\sigma\|s,$	$b\|\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$b\|\beta,$	$P + (b \rightarrow s))$	if s is a word
RIGHT	$(\sigma\|s'\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma\|s',$	$\beta,$	$P + (s' \rightarrow s))$	
REDUCE	$(\sigma\|s,$	$\beta,$	$P)$	$\Rightarrow$	$(\sigma,$	$\beta,$	$P)$	if s is a word

Table 13: ARCHYBRIDPARTIAL transition system.

SHIFT	$(\sigma\|s,\ b\|\beta)$	$\sigma \curvearrowright b$	b if h^*_b is in stack but not on top
	$(\sigma,\ \ b\|\beta)$	$\sigma \curvearrowleft b$	children of b that are in stack and unattached
LEFT		✗ $\curvearrowright s \curvearrowleft$ ✗	s if no h^*_s
	$(\sigma\|s,\ b\|\beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer but not on top
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\ \ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
RIGHT		✗ $\curvearrowright s \curvearrowleft$ ✗	s if no h^*_s
	$(\sigma\|s,\ \ \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s,\ \ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer
REDUCE	$(\sigma\|s,\ \ \beta)$	$s \curvearrowleft \beta$	s if h^*_s is in buffer
	$(\sigma\|s'\|s,\ \beta)$	$s' \curvearrowright s$	s if h^*_s is the second stack element
	$(\sigma\|s,\ \ \beta)$	$s \curvearrowright \beta$	children of s that are in buffer

Table 14: ARCHYBRIDPARTIAL dynamic cost: tokens that explicitly lose their head.

ARCHYBRIDPARTIAL applies the same ideas of partial parsing, but on the ARCHYBRID system. In this case, there are only three actions, but it is in fact not possible to encode empty attachments with three actions. So, we *extend* the transition set with an additional REDUCE action, which operates as in ARCEAGERPARTIAL: it pops the first stack element, even if it is unattached (which is necessarily the case here).

The resulting system consists in one shift action and three reduce actions, each one encoding a different attachment of s (s', b and empty). Compared to ARCEAGERPARTIAL, the semantics of each action are better singled out.

For the basic actions, the cost of ARCHYBRIDPARTIAL is the same as ARCHYBRID, augmented with enforcement of empty attachments (again, not in case of constraint-based filtering). The cost of REDUCE is the same as the basic RIGHT, except that it also penalizes reducing tokens whose head is on the left (i.e. when a RIGHT is required).

A.5.7 Short-spanned dependencies

Finally, we also propose transition systems that only learn and predict the shortest dependencies. These systems, ARCEAGERSPAN and ARCHYBRIDSPAN, are based on the partial systems ARCEAGERPARTIAL and ARCHYBRIDPARTIAL, and parameterized by a maximum dependency length (at parser initialization time). For instance, if the maximum length is 1, the parser will only predict attachments to neighbouring tokens, leaving unattached the tokens that typically have long distance attachments in the training data.

The constraint on dependency length only affects the action preconditions by forbidding LEFT and RIGHT actions which would create too long dependencies. Otherwise, it is completely identical to the partial systems. The cost is also the same, because long dependencies are filtered out from the reference as a preprocessing step, thus resulting in partial annotations containing only short dependencies.

B Global training strategies

In this appendix, we document the algorithms used by PanParser to implement global training in practice, which result from recently published ideas.

The main approach adopted in PanParser is indeed to consider all training strategies (including static oracles, local training and all variants of the main strategies) as special cases of global dynamic training.

B.1 Unified framework for local and global training

Algorithm 1 describes at a high level the workflow of global training: the sentence is initialized, an oracle (or strategy) component searches for an update pair, and the update is performed.

With Aufrant et al. (2017)'s proposal to restart training after the first update, the workflow now closely resembles that of local training. The only difference is that for local training, c^+ and c^- must be direct successors of c, while in global training they can be any descendant.

Algorithm 1: Global training on one sentence, with and without restart.

```
θ: model parameters, initialized to θ_0 before training
FINAL(·): true iff the whole sentence is processed
Function STRUCTUREDTRAINING(x,y)
    c ← INITIAL(x)
    c+, c− ← ORACLE(c, y, θ)
    θ ← UPDATE(θ, c+, c−)
Function STRUCTUREDTRAININGRESTART(x,y)
    c ← INITIAL(x)
    while ¬FINAL(c) do
        c+, c− ← ORACLE(c, y, θ)
        θ ← UPDATE(θ, c+, c−)
        c ← c+
```

B.2 Global dynamic oracle

PanParser training is based on the concept of global dynamic oracles (Aufrant et al., 2017), which is a direct extension of usual dynamic oracles to global training.

Similarly to local dynamic oracles which deem incorrect the *actions* which introduce a new error into the final parse, global dynamic oracles deem incorrect the *transition sequences* which introduce a new error into the final parse. Hence, given an initial configuration (not necessarily empty or gold), the correct configurations are those from which the maximum UAS is the same as the initial maximum.

The Boolean function that tests this condition, denoted $\textsc{Correct}_y(c'|c)$, can thus be efficiently computed using the COST function: a configuration c' is considered as CORRECT in the context of a configuration c, if there exists a sequence of transitions $t_1, \ldots, t_n$ such that $c' = c \circ t_1 \circ \cdots \circ t_n$ and $\textsc{Cost}(c, t_1) = \textsc{Cost}(c \circ t_1, t_2) = \ldots = \textsc{Cost}(c \circ \cdots \circ t_{n-1}, t_n) = 0$.

In other words, PanParser does not need to compute reference derivations explicitly, it just has to check the cost of each action it performs, and track the hypotheses that are still correct and those which are not.

Algorithm 2 shows how CORRECT is used to apply the early update and max-violation strategies with dynamic oracles.

B.3 Generic update strategy for structured training

Finally, since Algorithm 1 emphasizes on updates being the basic unit of all training procedures, Algorithm 3 takes a step back to show how all update strategies can be unified under the same generic framework.

Algorithm 2: Global dynamic oracle: error criterion and choice of an update configuration pair.

```
c_0: configuration to start decoding from
top_θ(·): best scoring element according to θ
NEXT(c): the set of all successors of c (or only c if it is final)
Function FINDVIOLATION(c_0, y, θ)
    Beam ← {c_0}
    while ∃c ∈ Beam, ¬FINAL(c) do
        Succ ← ∪_{c∈Beam} NEXT(c)
        Beam ← k-best(Succ, θ)
        if ∀c ∈ Beam, ¬CORRECT_y(c|c_0) then
            gold ← {c ∈ Succ | CORRECT_y(c|c_0)}
            return gold, Beam
    gold ← {c ∈ Beam | CORRECT_y(c|c_0)}
    return gold, Beam
Function EARLYUPDATEORACLE(c_0, y, θ)
    gold, Beam ← FINDVIOLATION(c_0, y, θ)
    return top_θ(gold), top_θ(Beam)
Function MAXVIOLATIONORACLE(c_0, y, θ)
    gold, Beam ← FINDVIOLATION(c_0, y, θ)
    candidates ← {(top_θ(gold), top_θ(Beam))}
    while ∃c ∈ Beam, ¬FINAL(c) do
        Succ ← ∪_{c∈Beam} NEXT(c)
        Beam ← k-best(Succ, θ)
        Succ+ ← ∪_{c∈gold} {c' ∈ NEXT(c) | CORRECT_y(c'|c_0)}
        gold ← k-best(Succ+, θ)
        candidates ← candidates + (top_θ(gold), top_θ(Beam))
    return arg max_{c+,c-∈candidates} (score_θ(c-) − score_θ(c+))
```

Note that in all state-of-the-art strategies, $\lambda_i = 1$, and that $k = 1$ for greedy training. Besides, under this framework the early update and max-violation strategies only differ in the choice of i_{update}, which is what is actually modeled in their PanParser implementation.

We believe that this framework faithfully represents the diversity of update strategies that can be envisioned in PanParser; it has been published in (Aufrant et al., 2016).

Algorithm 3: Generic update framework.

```
At time t_0: B_0 = {c_1, c_2, ..., c_k'}, k' ≤ k
k: maximum beam size
Function UPDATE(B_0, y, θ)
    B_1, B_2, ..., B_N ← DECODE(B_0, θ^{t_0}, k) such that FINAL(B_N)
    if ¬CORRECT_y(top(B_N)|B_0) then
        i_0 ← index of the first error detection (in B_{i_0})
        The algorithm chooses in turn:
        • using {B_i}_i, i_0, θ^{t_0}, CORRECT_y: a positive configuration c+ for each derivation length
        • using {B_i}_i, i_0, θ^{t_0}, {c+}: a negative configuration c- for each derivation length
        • using {B_i}_i, i_0, θ^{t_0}, {c+}, {c-}: a derivation length i_update
        c+_{i_update} = c_empty ∘ t+_0 ∘ ··· ∘ t+_{i_update} and c-_{i_update} = c_empty ∘ t-_0 ∘ ··· ∘ t-_{i_update}
        θ^{t_0+1} ← θ^{t_0} + Σ_{i=0}^{i_update} λ_i [φ(t+_i)) − φ(t-_i)]
```