PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 111 OCTOBER 2018 57–86

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

Lauriane Aufrant,[a][b] Guillaume Wisniewski[b]

[a] DGA, 60 boulevard du Général Martial Valin, 75 509 Paris, France
[b] LIMSI, CNRS, Univ. Paris-Sud, Université Paris-Saclay, 91 405 Orsay, France

Abstract

We present PanParser, a Python framework dedicated to transition-based structured prediction, and notably suitable for dependency parsing. On top of providing an easy way to train state-of-the-art parsers, as empirically validated on UD 2.0, PanParser is especially useful for research purposes: its modular architecture enables to implement most state-of-the-art transition-based methods under the same unified framework (out of which several are already built-in), which facilitates fair benchmarking and allows for an exhaustive exploration of slight variants of those methods. PanParser additionally includes a number of fine-grained evaluation utilities, which have already been successfully leveraged in several past studies, to perform extensive error analysis of monolingual as well as cross-lingual parsing.

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, like MaltParser (Nivre et al., 2006a), which is the reference implementation for transition-based parsers, and UDPipe (Straka and Straková, 2017), a popular pipeline system for neural parsing. In all transition-based parsers, the same elements are systematically found (a transition system, a classifier, an update procedure, etc.), corresponding to distinct lines of research. However, their implementations often adopt an ad-hoc architecture or a specific variation of each of these

Corresponding author: lauriane.aufrant@limsi.fr

Cite as: Lauriane Aufrant, Guillaume Wisniewski. PanParser: a Modular Implementation for Efficient Transition-Based Dependency Parsing. The Prague Bulletin of Mathematical Linguistics No. 111, 2018, pp. 57–86.
doi: 10.2478/pralin-2018-0007.

components, which impedes fair benchmarking and makes it difficult to evaluate the impact of a given component.

The PanParser framework aims at alleviating this effect by providing a modular architecture, in which most state-of-the-art transition-based 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. It is possible, for instance, to train a greedy neural ArcEager parser with a static oracle, or a delexicalized beam ArcHybrid parser with a dynamic oracle and an averaged perceptron.[1] PanParser also implements extensive utilities for error analysis. PanParser has been used, for instance, in cross-lingual transfer experiments (Aufrant et al., 2016b; Lacroix et al., 2016) or to design new learning strategies for dependency parsers (Aufrant et al., 2017).

PanParser differs from the other parsing frameworks by the ability to combine more freely alternate versions of each part of the parser, but also by the diversity of the built-in algorithms: for instance, UDPipe does not include the ArcEager and ArcHybrid transition systems and it lacks the support of global training strategies, spaCy[2] only offers ArcEager parsers trained with the max-violation strategy, while SyntaxNet (Andor et al., 2016) and the StanfordParser (Chen and Manning, 2014) focus on the ArcStandard system; as for MaltParser, it supports a large number of transition systems but neither global training nor dynamic oracles. PanParser also includes functionalities that are not found elsewhere, like newly-derived dynamic oracles or the ability to train projective parsers directly on non-projective data. On the other hand, our framework is not designed for pipelining as others are; the current built-ins also lack several non-projective transition systems included in UDPipe and MaltParser, although the architecture is already designed to support them – and recent works like Fernández-González and Gómez-Rodríguez (2018a)'s will help their implementation in future work. As for graph-based parsing strategies, like the seminal MSTParser (McDonald et al., 2005) or state-of-the-art neural ones (Kiperwasser and Goldberg, 2016; Dozat and Manning, 2017), they remain out of the scope of this framework.

The whole software 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. The core framework of PanParser can also be reused to implement other structured prediction tasks, as done for the built-in PoS tagger.

The rest of the paper is organized as follows. Section 2 presents the state of the art in transition-based dependency parsing, along the six lines of research which have

[1]See Section 2 and Supplementary A for a brief description of those algorithms.

[2]`https://spacy.io`

guided the design of PanParser. The main features of PanParser are described in Section 3 and Section 4 presents the scores achieved on the 73 treebanks of UD 2.0, for various configurations of PanParser. Further algorithmic and technical details are provided in the appendices and supplementary material: the procedure used to implement dynamic oracles for various transition systems in PanParser (Appendix A), the global dynamic oracle framework on which PanParser is based (Appendix B), as well as the built-in transition systems with their oracles (Supplementary A), practical examples of PanParser usage (Supplementary B) and a brief overview of the code architecture chosen to ensure modularity (Supplementary C).

PanParser is published under a BSD license and can be freely downloaded at `https://perso.limsi.fr/aufrant`.

2. Transition-based dependency parsing

Dependency parsing (Kübler et al., 2009) 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 denote syntactic dependencies between two words; in labeled parsing, a relation label is additionally assigned to each token. In this section, we introduce one class of algorithms building dependency trees, the transition-based approach (Nivre, 2008), which is the one adopted by PanParser.

In a transition-based parser, a parse is computed by performing a sequence of *transitions*, building the parse tree in an incremental fashion. The *parser configuration* thus represents a partially built dependency tree, and applying transition t to configuration c results in the parser moving to a *successor* configuration of c (denoted $c \circ t$), together with side effects on its internal state (typically based on stacks and lists): moving a token, creating an edge, etc.

According to Nivre (2008), '*a deterministic classifier-based parser consists of three essential components: a parsing algorithm, which defines the derivation of a syntactic analysis as a sequence of elementary parsing actions; a feature model, which defines a feature vector representation of the parser state at any given time; and a classifier, which maps parser states, as represented by the feature model, to parsing actions*'. Over the years, much work has been dedicated to improving parsers along those three lines: designing new parsing algorithms (i.e. transition systems, in the case of transition-based parsing) with various properties (Nivre, 2004, 2009; Gómez-Rodríguez and Nivre, 2010; Kuhlmann et al., 2011; Qi and Manning, 2017), more informative feature representations (Zhang and Clark, 2008; Zhang and Nivre, 2011; Bohnet et al., 2013; Alberti et al., 2015), and adapting the implementations to use more accurate classifiers (Chen and Manning, 2014; Dyer et al., 2015; Zhou et al., 2015; Andor et al., 2016).

However, in recent years, a series of contributions has been made in the way those three components interact, in particular at training time (Collins and Roark, 2004; Zhang and Clark, 2008; Goldberg and Nivre, 2012; Huang et al., 2012, Zhang and Nivre, 2012, Aufrant et al., 2017). The resulting lines of research have produced a number of algorithmic variants, with such diversity that we now find it beneficial to model these aspects as separate components. Therefore, the new reading grid we propose, and which has been adopted as the architecture of PanParser, is the following: a transition-based parser consists of 6 components (a transition system, a classifier, a feature model, a search component, an oracle and a training strategy), whose interactions can be controlled by a generic structured prediction framework, with no specific tie to the chosen algorithm or task.

In the following, we describe the state-of-the-art methods corresponding to each component, out of which most are or can be implemented under the PanParser framework. The actual built-ins will be listed in the next section, while the other algorithms can still be implemented within that framework, thanks to its modular architecture.

2.1. Transition system

The transition system defines the semantics of the actions predicted by the classifier, in order to relate them to actual dependency trees. Formally, it consists in four elements: the structure of the parser state (e.g. a stack and a buffer), a set of actions (e.g. Shift, Left, Right and Reduce), the semantics of each one (e.g. the Left action attaches the current token to another token on the right, and then discards it from the stack) and their preconditions (e.g. the Left action is invalid if the current token has already been attached).

Several such systems have been proposed in the literature, the most widely used ones being stack-based. Notably, ArcStandard (Nivre, 2003), ArcEager (Nivre, 2004) and ArcHybrid (Kuhlmann et al., 2011) rely on a stack (the currently processed tokens) and a buffer (the not yet processed tokens). They all consider candidate edges among the top few tokens on the stack and the first few tokens in the buffer, but they differ in the order in which edges are created, and regarding spurious ambiguity (i.e. whether the same tree can be produced by different derivations). All three systems guarantee parsing in linear time, but they are also subject to restrictions in expressivity (as they cannot produce non-projective trees, i.e. trees with crossing edges).

In order to improve expressivity or to facilitate the prediction of some edges, a number of other systems have been designed, using various kinds of internal state: a stack and a list (Nivre, 2009; Fernández-González and Gómez-Rodríguez, 2012), two lists and a buffer (Covington, 2001; Choi and Palmer, 2011), a single list (Goldberg and Elhadad, 2010), two (or more) stacks and a buffer (Gómez-Rodríguez and Nivre, 2010; Gómez-Rodríguez and Nivre, 2013), extra registers on top of the stack and the buffer (Pitler and McDonald, 2015), etc.

Many of those systems have seen additional variants, adding for instance non-monotonicity (Honnibal et al., 2013; Honnibal and Johnson, 2015; Fernández-González and Gómez-Rodríguez, 2017) to improve the robustness to erroneous predictions, or non-local transitions (Qi and Manning, 2017, Fernández-González and Gómez-

Rodríguez, 2018b), effectively shortening the derivations by collapsing series of Shift or Reduce actions into edge-creating transitions. The ability to parse given specific constraints on the output has also been investigated as a property of the transition system (Nivre and Fernández-González, 2014; Nivre et al., 2014).

Finally, several variants of each of those systems exist natively, depending on how the root tokens are handled. Indeed, the roots are supposed to remain unattached at the end of the derivation, but in order to alleviate boundary effects they are in general attached to a dummy Root token placed at the beginning or the end of the sentence. Yet, the position of this node can actually impact the semantics of the transition system, and it has been shown to impact the parsing accuracy (Ballesteros and Nivre, 2013).

2.2. Classifier

At each step of the parsing process, every possible transition is scored by a classifier, based on a feature representation of the current configuration, and the transitions to apply are chosen accordingly. This scoring step can be handled by any multi-class classifier, and several options have been envisioned in the literature.

Linear models have notably proved successful: Collins and Roark (2004) use an averaged perceptron – meaning that the final parameters retained after training are obtained by averaging over all values taken by the model throughout training – which has long remained a *de facto* standard (Huang et al., 2012). Another widespread strategy is to use support vector machines (Nivre et al., 2006b). A number of other techniques have been considered, like memory-based learning (Nivre et al., 2004), robust risk minimization (Choi and Nicolov, 2009) and confidence-weighted classifiers (Haulrich, 2010), but the largest body of recent work on the topic concerns the integration of neural networks, with various architectures: feedforward neural networks (Chen and Manning, 2014; Zhou et al., 2015), later augmented by a structured perceptron (Weiss et al., 2015) or a CRF loss (Andor et al., 2016), as well as recurrent networks (Stenetorp, 2013; Chen et al., 2015; Dyer et al., 2015). However, recurrent neural networks (and notably, LSTMs) cannot be implemented in the current version of the PanParser framework, which supports only stateless classifiers.

2.3. Feature model

The feature model specifies how the parser configuration is represented when it is fed to the classifier. Such features are typically extracted from 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, richer features like sequential (token distance), syntactic (valency, labels, representation of subtrees) or semantic information (semantic classes, pre-trained word embeddings) can also be collected for any such token (Zhang and Nivre, 2011; Agirre et al., 2011; Chen and Manning, 2014; Dyer et al., 2015).

	Standard templates
1 word	w, p and wp for S_0, N_0, N_1, N_2
2 words	$wp{\cdot}wp$, $wp{\cdot}w$, $w{\cdot}wp$, $wp{\cdot}p$, $p{\cdot}wp$, $w{\cdot}w$ and $p{\cdot}p$ for $S_0{\cdot}N_0$; $N_0p{\cdot}N_1p$
3 words	$p{\cdot}p{\cdot}p$ for $N_0{\cdot}N_1{\cdot}N_2$, $S_0{\cdot}N_0{\cdot}N_1$, $S_{0h}{\cdot}S_0{\cdot}N_0$, $S_0{\cdot}S_{0l}{\cdot}N_0$, $S_0{\cdot}S_{0r}{\cdot}N_0$, $S_0{\cdot}N_0{\cdot}N_{0l}$
	New templates with rich non-local features
Distance	$S_0w{\cdot}d$, $S_0p{\cdot}d$, $N_0w{\cdot}d$, $N_0p{\cdot}d$; $S_0w{\cdot}N_0w{\cdot}d$, $S_0p{\cdot}N_0p{\cdot}d$
Valency	S_0wv_l, S_0pv_l, S_0wv_r, S_0pv_r, N_0wv_l, N_0pv_l
Unigrams	w and p for S_{0h}, S_{0l}, S_{0r}, N_{0l}; l for S_0, S_{0l}, S_{0r}, N_{0l}
Third-order	w and p for S_{0h2}, S_{0l2}, S_{0r2}, N_{0l2}; l for S_{0h}, S_{0l2}, S_{0r2}, N_{0l2}; $p{\cdot}p{\cdot}p$ for $S_0{\cdot}S_{0h}{\cdot}S_{0h2}$, $S_0{\cdot}S_{0l}{\cdot}S_{0l2}$, $S_0{\cdot}S_{0r}{\cdot}S_{0r2}$, $N_0{\cdot}N_{0l}{\cdot}N_{0l2}$
Label set	S_0ws_l, S_0ps_l, S_0ws_r, S_0ps_r, N_0ws_l, N_0ps_l

Table 1: Feature templates proposed by Zhang and Nivre (2011). S_0 is the top stack element, N_0-N_2 the 3 first buffer elements. w, p and l stand for word, PoS tag and label. d is the token distance from S_0 to N_0. v_l (v_r) is the number of left (right) children, whose labels are s_l (s_r). $_h$ and $_{h2}$ denote the head and its own head, $_l$, $_{l2}$ ($_r$, $_{r2}$) the first and second leftmost (rightmost) children.

The resulting feature vector representation can be directly fed to non-linear classifiers like kernel SVMs and neural networks, while for linear models *feature templates* are typically used to combine the extracted atomic features into tuple features. Table 1 describes the templates handcrafted by Zhang and Nivre (2011), which are known for achieving state-of-the-art performance on several languages, while others advocate for automatic selection (Nilsson and Nugues, 2010; Ballesteros and Nivre, 2012; Ballesteros and Bohnet, 2014).

It can however be noted that depending on the exact parser settings, not all this information can be successfully extracted. For instance the delexicalized parsers used in cross-lingual transfer (Zeman and Resnik, 2008) make no use of wordforms, unlabeled parsers exclude label information, and depending on the transition system, some syntactic features remain unknown at prediction time (in bottom-up systems like ArcStandard and ArcHybrid the head of stack elements is never known, while in ArcEager it can be).

2.4. Search

Both during training and prediction, parsing is done by exploring the search space composed by all transitions scored by the classifier, in search for the best possible tree. Transition sequences are scored by summing the scores of all their transitions; parsing thus amounts to finding the derivation having the highest score. However, this inference step is hindered by the exponential size of the search space.

While exact inference can be done through dynamic programming (Huang and Sagae, 2010; Kuhlmann et al., 2011; Zhao et al., 2013), such methods imply severe restrictions on the set of authorized features, which jeopardize the accuracy of the parser. Instead, inexact search is generally employed, either as greedy or beam search. Greedy decoding (i.e. always following the single-best transition) is faster, but beam search (Zhang and Clark, 2008) yields more accurate parsers as it explores a larger part of the space: it maintains a set of k parse candidates (the beam) 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. This method faces some computational issues due to memory management, but they can be easily avoided by using tree-structured stacks and distributed representations of trees (Goldberg et al., 2013).

While their implementations often differ, it can be noted that greedy search is mathematically equivalent to beam search with $k = 1$. As for exhaustive search, it can be modeled with a beam of infinite size.

2.5. Oracle

Training of transition-based parsers is a two-step procedure: first some decoding is performed (using the chosen search strategy), then whenever an error a detected, the model is updated based on one (or more) predicted configuration(s) and one (or more) gold configuration(s).

The oracle is the component that governs the distinction between gold and erroneous configurations. More precisely, its role at training time is twofold: flagging errors during decoding and identifying gold configurations that can serve as positive (reference) configurations for updates, while the actual choice of update configurations depends on the training strategy.

The traditional and most straightforward kind of oracle is the static one. In a static oracle, references are precomputed heuristically, based on the semantics of the transition system: for each reference tree used as example, a unique derivation leading to that tree is chosen and its transitions become references, while all others are considered erroneous. While simple to implement, static oracles have two drawbacks: they ignore spurious ambiguity (when several derivations lead to the same reference tree, all but one are considered erroneous) and they are only defined along the reference derivation (given an arbitrary configuration in the search space, the optimal action to take next is not specified).

Instead, Goldberg and Nivre (2012) introduce dynamic oracles, defined as oracles that are both non-deterministic and complete: with dynamic oracles, the reference actions are tailored to the current configuration. In that framework, erroneous actions are defined as actions that introduce errors, i.e. that reduce the maximum score achievable on the current sentence (for instance by misattaching a token, or by removing from the stack a word which has not received all its children yet). The computation of the oracle is based on the *action cost*, which is a property of the transition system and numbers the errors introduced by each action, given an arbitrary configuration. The gold actions are defined as zero-cost actions; this way the oracle is guaranteed to find the best action(s) to perform, and it natively accounts for spurious ambiguity (an action leading to the reference tree is by design zero-cost) and is always defined (by definition of the maximum there is always at least one zero-cost action).

The main difficulty when applying dynamic oracles is to derive the action cost for the transition system, which requires a thorough investigation of its properties. So far, such oracles have been derived for ArcEager (Goldberg and Nivre, 2012), ArcHybrid and EasyFirst (Goldberg and Nivre, 2013), ArcStandard (Goldberg et al., 2014), as well as several non-monotonic (Honnibal et al., 2013; Honnibal and Johnson, 2015) and non-projective systems (Gómez-Rodríguez et al., 2014; Gómez-Rodríguez and Fernández-González, 2015; Fernández-González and Gómez-Rodríguez, 2017). A few other works have proposed oracles drawing from this line of research but which are only partly dynamic, incomplete or approximated (Björkelund and Nivre, 2015; de Lhoneux et al., 2017; Fernández-González and Gómez-Rodríguez, 2018a). Aufrant et al. (2018) additionally propose a systematic way to approximate the oracle when it has not yet been derived to handle all particular cases – for instance for reference trees that are out of the expressive scope of the transition system, typically non-projective training examples for projective parsers – by using an unsound cost and considering the minimum-cost actions as gold. This is the approach adopted within our framework; it is further described in Appendix A, which explains in which measure PanParser departs from the standard dynamic oracle framework.

While dynamic oracles are primarily defined to identify reference *actions*, Aufrant et al. (2017) extend them to reference *transition sequences*, to accommodate their use with beam parsers. The advantage of these *global dynamic oracles* is that they do not require any explicit computation of the reference set (which can be exponentially huge), as they can be implemented by simply tracking the action costs inside the beam.

2.6. Training strategy

The literature traditionally distinguishes local and global training strategies. In local training, each decision is optimized independently: all along the derivation, the action predicted by the classifier is checked against the gold action(s) (the reference(s) provided by the oracle), and an update occurs whenever they differ.

When beam search is employed, local training is suboptimal (Zhang and Nivre, 2012) and a global criterion is preferred, meaning that the parameters are updated once for each training sentence, depending on the optimality of transition *sequences*. Algorithm 1 (left part) summarizes the training for each sentence x (with gold parse y): $\textsc{Initial}(x)$ denotes the initial configuration for x and the $\textsc{Oracle}$ procedure performs decoding to find configurations that play the role of the 'positive' and 'negative' examples (resp. c^+ and c^-) required by the $\textsc{Update}$ operation (typically a perceptron update rule (Collins and Roark, 2004) or a gradient computation with the globally normalized loss of Andor et al. (2016)). Several strategies, corresponding to various implementations of the ORACLE function, have been used to find these examples.

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

```
$\theta$: model parameters, initialized to $\theta_0$ before training
FINAL($\cdot$): true iff the whole sentence is processed

Function TrainOneSentence($x$,$y$)
    $c \leftarrow$ INITIAL($x$)
    $c^+, c^- \leftarrow$ ORACLE($c$, $y$, $\theta$)
    $\theta \leftarrow$ UPDATE($\theta$, $c^+$, $c^-$)

Function TrainOneSentenceRestart($x$,$y$)
    $c \leftarrow$ INITIAL($x$)
    while $\neg$FINAL($c$) do
        $c^+, c^- \leftarrow$ ORACLE($c$, $y$, $\theta$)
        $\theta \leftarrow$ UPDATE($\theta$, $c^+$, $c^-$)
        $c \leftarrow c^+$
```

In the EarlyUpdate strategy (Collins and Roark, 2004; Zhang and Clark, 2008), the sentence is parsed using conventional beam decoding, checking at each step whether the reference tree is still reachable, and an update happens as soon as the reference derivation (or all references, depending on what the oracle generates) falls off the beam: the top scoring configuration at this step is penalized and the reference that has just fallen off the beam is reinforced. Another strategy, MaxViolation (Huang et al., 2012), is to continue decoding even though the reference has fallen off the beam, in order to find the configuration having the largest gap between the scores of the (partial) hypothesis and the (partial) gold derivation. Compared to EarlyUpdate, MaxViolation speeds up convergence by covering longer transition sequences and can yield slightly better parsers.

In the standard version of these strategies, after a global update the rest of the sequence is ignored, moving on to the next example. However, Aufrant et al. (2017) extend those strategies with a restart option (right part of Algorithm 1) which reinitializes the beam after each update and enables further updates on the same example, so that the whole sentence is exploited during training, with benefits in terms of convergence, accuracy and sampling distributions.

Notably, under the restart framework – and similarly to the equivalence of search strategies – local training can be interpreted as a special case of global training, when applying early update and restart on a beam of size 1.

One key aspect when choosing a training strategy is how the training configurations are generated: each update (either local or global) raises questions on which configuration to restart from, the positive or the negative one. Goldberg and Nivre (2012) show that error exploration, i.e. pursuing on the erroneous path, improves the accuracy by making the classifier able to produce the next best tree, even when the optimal one has become unreachable. Classifiers that stick to the gold space at training time suffer indeed from error propagation, as the suboptimal configurations they are confronted to at prediction time are an unknown territory in which they were never trained to take good decisions. As a trade-off between strict supervision and robustness, various exploration policies can be envisioned; for instance Goldberg and Nivre (2013) keep the first iteration in the gold space, and then apply error exploration with a probability of 90%, while Ballesteros et al. (2016) sample the next configuration from the probability distribution output by the classifier. However, error exploration requires oracle completeness, and can thus only be entertained when using a dynamic oracle.

3. The PanParser implementation

As described in the previous section, a transition-based parser can be viewed as the association of several components: a *transition system* (associating parse trees with transition sequences), a *classifier* (scoring transitions based on a feature representation), a *feature model* (extracting feature vectors from parse configurations), a *search strategy* (producing transition sequences, given a classifier model), an *oracle* (mapping gold annotations to gold transitions) and a *training strategy* (effectively choosing the training configurations to update the model).

In PanParser, to ensure modularity and compatibility, all of these components are implemented separately – and interfaced by a generic structured prediction framework, which handles the main training and prediction logic. The first three correspond to distinct modules, for which we provide several implementations, and which can be easily extended by alternate models or implementations. The definition of the *transition system* includes all properties that are system-specific, so that the other modules can be fully semantics-agnostic: its API can be queried to return (given a configuration) the list of valid actions, the action costs, a successor configuration after a given action, the partial tree already built (as an on-demand computation based on transition history), and whether the state is final. The *classifier* is also seen as a black box by the rest of the software, with a score/predict/update API (plus some initialization functions) that makes it possible to integrate any stateless classifier; support for stateful classifiers (like LSTMs) is not yet included but is planned for future work as an API extension. As for the *feature model*, it relies on high-level properties of the parse configurations (i^{th} word in buffer, head of the top of the stack, etc.) to generate feature representations, either as atomic or templated features (to accommodate both linear and non-linear classifiers).

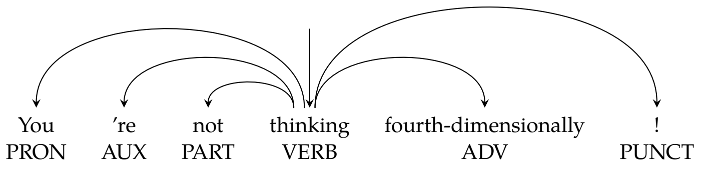
The three other components (search, oracle and training strategy) are based on an extensive set of built-in parameters which can be set and combined at will. Following Aufrant et al. (2016a), the whole search/learn procedure is based on beam search and global dynamic oracles, from which all other strategies (greedy search, static oracle, local training) are derived as special cases.

On top of algorithmic analyses, keeping those components independent has also required specific implementation choices and abstraction layers; more details on that matter are provided in Supplementary C.

After a brief description of how dependency trees are represented internally (§3.1), the built-in components of PanParser are described in §3.2. Further technical details are then provided on other functionalities of PanParser: enriched inputs (§3.3), parser ensembling (§3.4), a built-in PoS tagger (§3.5) and the error analysis tools (§3.6). See Supplementary B for an illustration of how these functionalities can be used and combined.

3.1. Representation of a dependency tree

Since there is a one-to-one correspondence between child tokens and dependency edges, a dependency tree can be easily modeled as a list of head tokens with padding elements (a dummy Root token or symbols denoting the start and end of sentences). In PanParser, it is represented as a list of integers, the integer at position i corresponding to the index of the head governing the i^{th} word in the sentence. Relation labels can similarly be represented as a list of strings, for labeled parsing. Figure 1 illustrates the resulting representation of a tree, with three variants depending on the position of the Root token (None, First and Last, as empirically compared by Ballesteros and Nivre (2013)); in PanParser the last position is used by default, but the first position can be set to be used internally – parsing without Root token is currently not supported.



	You^{0}	're^{1}	not^{2}	thinking^{3}	fourth-dimensionally^{4}	!^{5}	
	3	3	3	None	3	3	
ROOT^{0}	You^{1}	're^{2}	not^{3}	thinking^{4}	fourth-dimensionally^{5}	!^{6}	
None	4	4	4	0	4	4	
<start>^{0}	You^{1}	're^{2}	not^{3}	thinking^{4}	fourth-dimensionally^{5}	!^{6}	ROOT^{7}
None	4	4	4	7	4	4	0
None	nsubj	aux	advmod	root	advmod	punct	None

Figure 1: Dependency tree representations with various Root positions.

3.2. Built-in components

Transition system PanParser implements four transition systems (ArcEager, ArcHybrid, ArcStandard and NonMonotonicArcEager) as well as variants for partial output and short-spanned dependencies. For all, both versions with the Root token in leading and trailing positions are implemented. Some experimental options are also included, like adding head direction constraints to ArcEager.

High-level functions (extracting all possible atomic features, enumerating stack tokens, etc.) are available to facilitate the implementation of other transition systems based on a stack and a buffer; for other state structures they remain to be implemented.

A formal description of all built-in systems and their action costs is provided in Supplementary A. For each one, the soundness of action costs has been experimentally validated by exhaustive search from all configurations on all possible trees for sentences under 10 tokens.

Classifier The default classifier used by PanParser is the multi-class averaged (structured) perceptron, following Collins and Roark (2004). Support for neural networks is also included, and a vanilla feedforward neural network (based on Keras and Theano) is implemented as a proof of concept.

Additionally, generic classifiers are provided for joint prediction and voting, which are in fact wrappers around other classifiers. Joint classifiers are natively used for instance to enable labeled parsing: based on the same feature representations, actions and relation labels are predicted in parallel by two classifiers (which may or may not share parameters, depending on the implemented classifiers). As for the voting wrapper, it enables parser ensembling with (weighted) votes at the action level.

Feature model PanParser is shipped with the feature templates of Zhang and Nivre (2011), together with options that extend them with morphological features and pre-trained embeddings; it is also possible to use the atomic version of the features (for non-linear classifiers) or to write custom templates, based on the provided atomic features (including both coarse and fine-grained PoS tags). In any case, two global parameters control whether lexicalized features and label information are included (to build delexicalized and unlabeled parsers).

Search As explained above, PanParser allows both greedy and beam search, the former being a special case of the latter; it consequently implements the data structure introduced for beam search by Goldberg et al. (2013), based on immutable objects and distributed representations. This implementation is, however, suboptimal for vanilla greedy parsers, which can be optimized using mutable objects, so that we also provide an alternate implementation dedicated to greedy parser states.

The mutable version represents the state of the parser as a buffer pointer, a stack and a parse tree, that get updated whenever a transition is applied. This structure makes much information available in constant time (which notably speeds up feature extraction), but it cannot be used for beam search without costly object copies (Goldberg et al., 2013) and has consequently limited functionalities. The mathematical equivalence of both implementations for greedy parsing has been experimentally validated.

In Goldberg et al. (2013)'s version, which is the default, 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).[3] 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, factoring information in this way makes beam search and global training cheaper, both in time and memory usage.

Oracle Both static and dynamic oracles (including global dynamic ones) are supported by PanParser; static oracles are actually computed using dynamic ones – the reference derivation is built by pre-parsing the sentence while restricting the search space to zero-cost actions and ignoring their score.

Training strategy PanParser supports both local and global training, with several strategies: early update, max-violation, full update (even though Huang et al. (2012) discourage its use), several variants of those (e.g. to optimize the similarity of the positive and negative configurations), together with the restart option and arbitrary exploration policies.

As explained in §2.6, local and global training are unified under the same framework (Algorithm 1), 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, and iterate until the example is considered processed.

The various update strategies for global training can themselves be unified under the same framework, as shown by Algorithm 2 (Correct being the error criterion used by global dynamic oracles): with this viewpoint, EarlyUpdate and MaxViolation only differ in the choice of i_{update}, which is actually how they are implemented in

[3]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.

PanParser (with lazy expansion of the beam and anticipation of errors). Appendix B further describes how those strategies fit within the dynamic oracle framework.

Algorithm 2: Basic scheme for training strategies (Aufrant et al., 2016a).

```
At time $t_0$: $B_0 = \{c_1, c_2, ..., c_{k'}\}, k' \leq k$
$k$: maximum beam size (1 for local training)
$\lambda_i$: optional focus on early/late transitions (1 in all state-of-the-art strategies)
Function TrainingUnit($B_0$, $y$, $\theta$)
    $B_1, B_2, ..., B_N \leftarrow$ Decode($B_0, \theta^{t_0}, k$) such that Final($B_N$)
    if $\neg$Correct$_y$(top($B_N$)|$B_0$) then
        $i_0 \leftarrow$ index of the first error detection (in $B_{i_0}$)
        The algorithm chooses in turn:
        • using $\{B_i\}_i, i_0, \theta^{t_0}$, Correct$_y$: a positive configuration $c^+$ for each
          derivation length
        • using $\{B_i\}_i, i_0, \theta^{t_0}, \{c^+\}$: a negative configuration $c^-$ for each
          derivation length
        • using $\{B_i\}_i, i_0, \theta^{t_0}, \{c^+\}, \{c^-\}$: a derivation length $i_{update}$
        $c^+_{i_{update}} = c_{empty} \circ t^+_0 \circ \cdots \circ t^+_{i_{update}}$
        $c^-_{i_{update}} = c_{empty} \circ t^-_0 \circ \cdots \circ t^-_{i_{update}}$
        The global update is effected, e.g. with the perceptron rule:
        • $\theta^{t_0+1} \leftarrow \theta^{t_0} + \sum_{i=0}^{i_{update}} \lambda_i \left[\phi(t^+_i)) - \phi(t^-_i)\right]$
```

On top of the errors flagged by the oracle, PanParser also accepts other non-standard stopping criteria, like forcing the beam to reinitialize every few actions (thus preventing updates on very distant configurations).

3.3. Enriched input: partial trees and constraints

Compared to traditional implementations, PanParser makes an extensive use of the dynamic oracle framework to leverage partial trees in several ways: it supports training on partially annotated sentences (which enables robustness to incomplete datasets, but also fine-grained subsampling), predicting under partial constraints (when the head, or at least the dependency direction, is already provided for a few tokens) and training under constraints (for better train-test consistency, and using features extracted from the constraints). It is also possible, using the corresponding transition systems, to learn to predict partial trees directly – in the PanParser framework, training and prediction unfold as usual even for such partial parsers.

The reason why dynamic oracles enable training on examples with partial annotations is that they make the updates error-driven: 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). So, provided that the action costs are implemented appropriately (i.e. not assuming the existence of annotations), which is the case in PanParser, training on such data is possible by design. 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.

As for constrained training and prediction, they rely on a straightforward action filtering, based on dynamic oracles: 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.

3.4. Parser ensembling

Two strategies for parser ensembling are implemented under the PanParser framework: parser cascading (Aufrant and Wisniewski, 2017), which consists in pipelining a series of partial parsers, and MST-based reparsing (Sagae and Lavie, 2006). Reparsing enables a token-level vote on the output of several parsers; when weighting the contribution of each parser, which PanParser allows, this strategy can for instance be used in cross-lingual parsing, to combine various sources (Rosa and Žabokrtský, 2015).

3.5. Support for other structured prediction tasks: PoS tagging

PanParser also 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.

3.6. Error analysis

In order to facilitate extensive error analysis, PanParser is shipped with a series of evaluation tools, both for computing overall accuracies and fine-grained statistics. Several built-in criteria (PoS tags, dependency length, direction, position in sentence, word frequency, etc.) can be used (and combined) to guide the analysis or narrow the results. Examples of the studies enabled by PanParser are presented in Supplementary B.

4. Experiments

The accuracy of PanParser is evaluated on the 73 treebanks of the Universal Dependencies 2.0 (Nivre et al., 2017a,b). Table 2 reports the average scores achieved with the main few settings, together with similar measures for three other open source parsers: MaltParser 1.9 (Nivre et al., 2006a), MSTParser 0.5.1 (McDonald et al., 2005) and UDPipe 1.1 (Straka and Straková, 2017).[4]

Our system appears competitive with the other parsers, all of them being outperformed by an 8-sized beam PanParser. Further comparison with UDPipe reveals that both systems are actually on par on large treebanks (more than 500 sentences), while PanParser outperforms all parsers by a large margin on small treebanks (less than 500 sentences).

As a side note, the gains achieved by PanParser when changing the training strategy and Root position also appear consistent with the literature (Goldberg and Nivre, 2012; Zhang and Nivre, 2012; Huang et al., 2012; Ballesteros and Nivre, 2013), which validates previous results in this new framework.

5. 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 arbitrary user-defined atomic features. 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, e.g. 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).

[4]We use the default settings for MaltParser (ArcEager parser with a linear classifier and no pseudo-projectivization) and MSTParser (first-order projective parser), and Straka (2017)'s hyperparameters for UDPipe.

System	Root position	Greedy	Greedy dynamic	Early update	Max-violation
ArcEager	First	77.89	78.97	80.29	80.36
	Small ‖ Large	66.27 ‖ 81.15	68.17 ‖ 82.00	68.48 ‖ 83.60	68.42 ‖ 83.71
	Last	78.63	79.43	80.35	80.40
	Small ‖ Large	67.60 ‖ 81.72	68.70 ‖ 82.44	68.58 ‖ 83.66	68.87 ‖ 83.63
ArcHybrid	First	75.72	76.54	79.39	79.78
	Small ‖ Large	66.56 ‖ 78.29	66.49 ‖ 79.36	66.72 ‖ 82.95	68.43 ‖ 82.96
	Last	76.02	77.05	79.70	79.86
	Small ‖ Large	66.74 ‖ 78.62	67.42 ‖ 79.76	68.39 ‖ 82.87	68.61 ‖ 83.02
MaltParser		72.88 (58.87 ‖ 76.82)			
MSTParser		79.52 (65.59 ‖ 83.43)			
UDPipe		79.47 (64.48 ‖ 83.67)			

Table 2: Average UAS achieved by PanParser on UD 2.0 with various strategies, compared to several open source parsers. 'Greedy' results are computed with a static oracle, but for fair comparison of both oracles the non-projective examples are also exploited (using a precomputed reference approximated by a dynamic oracle). 'Greedy dynamic' chooses exploration after each update. The 'Early update' and 'Max-violation' strategies use global dynamic oracles without restart. 'Small' and 'Large' results distinguish the treebanks under and over 500 sentences. For fair comparison, UDPipe is trained without pre-trained embeddings, which would have significantly increased the available information.

Acknowledgments

This work has been partly funded by the French *Direction générale de l'armement* and by the *Agence Nationale de la Recherche* (ParSiTi project, ANR-16-CE33-0021).

Bibliography

Agirre, Eneko, Kepa Bengoetxea, Koldo Gojenola, and Joakim Nivre. Improving Dependency Parsing with Semantic Classes. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 699–703, Portland, Oregon, USA, 6 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-2123`.

Alberti, Chris, David Weiss, Greg Coppola, and Slav Petrov. Improved Transition-Based Parsing and Tagging with Neural Networks. In *Proceedings of the 2015 Conference on Empirical*

Methods in Natural Language Processing, pages 1354–1359, Lisbon, Portugal, 9 2015. Association for Computational Linguistics. URL `http://aclweb.org/anthology/D15-1159`.

Andor, Daniel, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally Normalized Transition-Based Neural Networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2442–2452, Berlin, Germany, 8 2016. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P16-1231`.

Aufrant, Lauriane and Guillaume Wisniewski. LIMSI@CoNLL'17: UD Shared Task. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 163–173, Vancouver, Canada, 8 2017. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/K17-3017`.

Aufrant, Lauriane, Guillaume Wisniewski, and François Yvon. 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, 2016a.

Aufrant, Lauriane, Guillaume Wisniewski, and François Yvon. Zero-resource Dependency Parsing: Boosting Delexicalized Cross-lingual Transfer with Linguistic Knowledge. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 119–130, Osaka, Japan, 12 2016b. The COLING 2016 Organizing Committee. URL `http://aclweb.org/anthology/C16-1012`.

Aufrant, Lauriane, Guillaume Wisniewski, and François Yvon. 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, Valencia, Spain, 4 2017. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/E17-2051`.

Aufrant, Lauriane, Guillaume Wisniewski, and François Yvon. Exploiting Dynamic Oracles to Train Projective Dependency Parsers on Non-Projective Trees. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 413–419, New Orleans, Louisiana, 6 2018. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N18-2066`.

Ballesteros, Miguel and Bernd Bohnet. Automatic Feature Selection for Agenda-Based Dependency Parsing. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 794–805, Dublin, Ireland, 8 2014. Dublin City University and Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/C14-1076`.

Ballesteros, Miguel and Joakim Nivre. MaltOptimizer: An Optimization Tool for MaltParser. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 58–62, Avignon, France, 4 2012. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/E12-2012`.

Ballesteros, Miguel and Joakim Nivre. Going to the Roots of Dependency Parsing. *Computational Linguistics*, 39(1):5–13, 2013. URL `http://www.aclweb.org/anthology/J/J13/J13-1002.pdf`.

Ballesteros, Miguel, Yoav Goldberg, Chris Dyer, and Noah A. Smith. Training with Exploration Improves a Greedy Stack LSTM Parser. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2005–2010, Austin, Texas, 11 2016. Association for Computational Linguistics. URL `https://aclweb.org/anthology/D16-1211`.

Björkelund, Anders and Joakim Nivre. Non-Deterministic Oracles for Unrestricted Non-Projective Transition-Based Dependency Parsing. In *Proceedings of the 14th International Conference on Parsing Technologies*, pages 76–86, Bilbao, Spain, 7 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W15-2210`.

Bohnet, Bernd, Joakim Nivre, Igor Boguslavsky, Richárd Farkas, Filip Ginter, and Jan Hajič. Joint morphological and syntactic analysis for richly inflected languages. *Transactions of the Association for Computational Linguistics*, 1:415–428, 2013.

Chen, Danqi and Christopher Manning. A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar, 10 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1082`.

Chen, Xinchi, Yaqian Zhou, Chenxi Zhu, Xipeng Qiu, and Xuanjing Huang. Transition-based Dependency Parsing Using Two Heterogeneous Gated Recursive Neural Networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1879–1889, Lisbon, Portugal, 9 2015. Association for Computational Linguistics. URL `http://aclweb.org/anthology/D15-1215`.

Choi, Jinho D and Nicolas Nicolov. K-best, locally pruned, transition-based dependency parsing using robust risk minimization. *Recent Advances in Natural Language Processing V*, 309: 205–216, 2009.

Choi, Jinho D. and Martha Palmer. Getting the Most out of Transition-based Dependency Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 687–692, Portland, Oregon, USA, 6 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-2121`.

Collins, Michael and Brian Roark. 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, Barcelona, Spain, 7 2004. doi: 10.3115/1218955.1218970. URL `http://www.aclweb.org/anthology/P04-1015`.

Covington, Michael A. A Fundamental Algorithm for Dependency Parsing. In *Proceedings of the 39th annual ACM southeast conference*, pages 95–102, 2001.

de Lhoneux, Miryam, Sara Stymne, and Joakim Nivre. Arc-Hybrid Non-Projective Dependency Parsing with a Static-Dynamic Oracle. In *Proceedings of the 15th International Conference on Parsing Technologies*, pages 99–104, Pisa, Italy, 9 2017. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W17-6314`.

Dozat, Timothy and Christopher D. Manning. Deep Biaffine Attention for Neural Dependency Parsing. *ICLR 2017*, 2017. URL `http://arxiv.org/abs/1611.01734`.

Dyer, Chris, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. Transition-Based Dependency Parsing with Stack Long Short-Term Memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint*

Conference on Natural Language Processing (Volume 1: Long Papers), pages 334–343, Beijing, China, 7 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-1033`.

Fernández-González, Daniel and Carlos Gómez-Rodríguez. Improving Transition-Based Dependency Parsing with Buffer Transitions. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 308–319, Jeju Island, Korea, 7 2012. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D12-1029`.

Fernández-González, Daniel and Carlos Gómez-Rodríguez. A Full Non-Monotonic Transition System for Unrestricted Non-Projective Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 288–298, Vancouver, Canada, 7 2017. Association for Computational Linguistics. URL `http://aclweb.org/anthology/P17-1027`.

Fernández-González, Daniel and Carlos Gómez-Rodríguez. A Dynamic Oracle for Linear-Time 2-Planar Dependency Parsing. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 386–392, New Orleans, Louisiana, 6 2018a. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N18-2062`.

Fernández-González, Daniel and Carlos Gómez-Rodríguez. Non-Projective Dependency Parsing with Non-Local Transitions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 693–700, New Orleans, Louisiana, 6 2018b. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N18-2109`.

Goldberg, Yoav and Michael Elhadad. An Efficient Algorithm for Easy-First Non-Directional Dependency Parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 742–750, Los Angeles, California, 6 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N10-1115`.

Goldberg, Yoav and Joakim Nivre. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India, 12 2012. The COLING 2012 Organizing Committee. URL `http://www.aclweb.org/anthology/C12-1059`.

Goldberg, Yoav and Joakim Nivre. Training Deterministic Parsers with Non-Deterministic Oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414, 2013. ISSN 2307-387X. URL `https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/145`.

Goldberg, Yoav, Kai Zhao, and Liang Huang. 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, Sofia, Bulgaria, 8 2013. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P13-2111`.

Goldberg, Yoav, Francesco Sartorio, and Giorgio Satta. A Tabular Method for Dynamic Oracles in Transition-Based Parsing. *Transactions of the Association for Computational Linguistics*, 2: 119–130, 2014. ISSN 2307-387X. URL `https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/302`.

Gómez-Rodríguez, Carlos and Daniel Fernández-González. 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, Beijing, China, 7 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-2042`.

Gómez-Rodríguez, Carlos and Joakim Nivre. A Transition-Based Parser for 2-Planar Dependency Structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1492–1501, Uppsala, Sweden, 7 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P10-1151`.

Gómez-Rodríguez, Carlos and Joakim Nivre. Divisible transition systems and multiplanar dependency parsing. *Computational Linguistics*, 39(4):799–845, 2013.

Gómez-Rodríguez, Carlos, Francesco Sartorio, and Giorgio Satta. A Polynomial-Time Dynamic Oracle for Non-Projective Dependency Parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 917–927, Doha, Qatar, 10 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1099`.

Haulrich, Martin. Transition-Based Parsing with Confidence-Weighted Classification. In *Proceedings of the ACL 2010 Student Research Workshop*, pages 55–60, Uppsala, Sweden, 7 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P10-3010`.

Honnibal, Matthew and Mark Johnson. An Improved Non-monotonic Transition System for Dependency Parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, 9 2015. Association for Computational Linguistics. URL `http://aclweb.org/anthology/D15-1162`.

Honnibal, Matthew, Yoav Goldberg, and Mark Johnson. A Non-Monotonic Arc-Eager Transition System for Dependency Parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172, Sofia, Bulgaria, 8 2013. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W13-3518`.

Huang, Liang and Kenji Sagae. Dynamic Programming for Linear-Time Incremental Parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086, Uppsala, Sweden, 7 2010. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P10-1110`.

Huang, Liang, Suphan Fayong, and Yang Guo. 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, Montréal, Canada, 6 2012. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N12-1015`.

Kiperwasser, Eliyahu and Yoav Goldberg. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations. *Transactions of the Association for Computational Linguistics*, 4:313–327, 2016. ISSN 2307-387X. URL `https://transacl.org/ojs/index.php/tacl/article/view/885`.

Kübler, Sandra, Ryan McDonald, and Joakim Nivre. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127, 2009.

Kuhlmann, Marco, Carlos Gómez-Rodríguez, and Giorgio Satta. 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, Portland, Oregon, USA, 6 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-1068`.

Lacroix, Ophélie, Lauriane Aufrant, Guillaume Wisniewski, and François Yvon. Frustratingly Easy Cross-Lingual Transfer for Transition-Based Dependency Parsing. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1058–1063, San Diego, California, 6 2016. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N16-1121`.

McDonald, Ryan, Fernando Pereira, Kiril Ribarov, and Jan Hajic. Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver, British Columbia, Canada, 10 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/H/H05/H05-1066`.

Nilsson, Peter and Pierre Nugues. Automatic Discovery of Feature Sets for Dependency Parsing. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 824–832, Beijing, China, 8 2010. Coling 2010 Organizing Committee. URL `http://www.aclweb.org/anthology/C10-1093`.

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

Nivre, Joakim. Incrementality in Deterministic Dependency Parsing. In Keller, Frank, Stephen Clark, Matthew Crocker, and Mark Steedman, editors, *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain, 7 2004. Association for Computational Linguistics.

Nivre, Joakim. Algorithms for Deterministic Incremental Dependency Parsing. *Comput. Linguist.*, 34(4):513–553, 2008. ISSN 0891-2017. doi: 10.1162/coli.07-056-R1-07-027. URL `http://dx.doi.org/10.1162/coli.07-056-R1-07-027`.

Nivre, Joakim. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore, 8 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P/P09/P09-1040`.

Nivre, Joakim and Daniel Fernández-González. Arc-eager parsing with the tree constraint. *Computational linguistics*, 40(2):259–267, 2014.

Nivre, Joakim, Johan Hall, and Jens Nilsson. Memory-Based Dependency Parsing. In Ng, Hwee Tou and Ellen Riloff, editors, *HLT-NAACL 2004 Workshop: Eighth Conference on Computational Natural Language Learning (CoNLL-2004)*, pages 49–56, Boston, Massachusetts, USA, May 6 - May 7 2004. Association for Computational Linguistics.

Nivre, Joakim, Johan Hall, and Jens Nilsson. MaltParser: A Data-Driven Parser-Generator for Dependency Parsing. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, volume 6, pages 2216–2219, 2006a.

Nivre, Joakim, Johan Hall, Jens Nilsson, Gülşen Eryiğit, and Svetoslav Marinov. Labeled Pseudo-Projective Dependency Parsing with Support Vector Machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 221–225, New York City, 6 2006b. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W06/W06-2933`.

Nivre, Joakim, Yoav Goldberg, and Ryan McDonald. Constrained arc-eager dependency parsing. *Computational Linguistics*, 40(2):249–527, 2014.

Nivre, Joakim, Željko Agić, Lars Ahrenberg, et al. Universal Dependencies 2.0, 2017a. URL `http://hdl.handle.net/11234/1-1983`. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague.

Nivre, Joakim, Željko Agić, Lars Ahrenberg, et al. Universal Dependencies 2.0 – CoNLL 2017 Shared Task Development and Test Data, 2017b. URL `http://hdl.handle.net/11234/1-2184`. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University.

Pitler, Emily and Ryan McDonald. A Linear-Time Transition System for Crossing Interval Trees. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 662–671, Denver, Colorado, May–June 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N15-1068`.

Qi, Peng and Christopher D. Manning. Arc-swift: A Novel Transition System for Dependency Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 110–117, Vancouver, Canada, 7 2017. Association for Computational Linguistics. URL `http://aclweb.org/anthology/P17-2018`.

Rosa, Rudolf and Zdeněk Žabokrtský. KLcpos3 - a Language Similarity Measure for Delexicalized Parser Transfer. 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 243–249, Beijing, China, 7 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-2040`.

Sagae, Kenji and Alon Lavie. Parser Combination by Reparsing. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, pages 129–132, New York City, USA, 6 2006. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N/N06/N06-2033`.

Stenetorp, Pontus. Transition-based dependency parsing using recursive neural networks. In *NIPS Workshop on Deep Learning*, 2013.

Straka, Milan. CoNLL 2017 Shared Task - UDPipe Baseline Models and Supplementary Materials, 2017. URL `http://hdl.handle.net/11234/1-1990`. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University.

Straka, Milan and Jana Straková. Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada, 8 2017. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/K17-3009`.

Swayamdipta, Swabha, Miguel Ballesteros, Chris Dyer, and Noah A. Smith. Greedy, Joint Syntactic-Semantic Parsing with Stack LSTMs. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 187–197, Berlin, Germany, 8 2016. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/K16-1019.

Weiss, David, Chris Alberti, Michael Collins, and Slav Petrov. Structured Training for Neural Network Transition-Based 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 1: Long Papers)*, pages 323–333, Beijing, China, 7 2015. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P15-1032.

Zeman, Daniel and Philip Resnik. Cross-Language Parser Adaptation between Related Languages. In *Proceedings of the IJCNLP-08 Workshop on NLP for Less Privileged Languages*, pages 35–42, 2008.

Zhang, Yue and Stephen Clark. 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, Honolulu, Hawaii, 10 2008. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/D08-1059.

Zhang, Yue and Joakim Nivre. 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, Portland, Oregon, USA, 6 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-2033.

Zhang, Yue and Joakim Nivre. Analyzing the Effect of Global Learning and Beam-Search on Transition-Based Dependency Parsing. In *Proceedings of COLING 2012: Posters*, pages 1391–1400, Mumbai, India, 12 2012. The COLING 2012 Organizing Committee. URL http://www.aclweb.org/anthology/C12-2136.

Zhao, Kai, James Cross, and Liang Huang. Optimal Incremental Parsing via Best-First Dynamic Programming. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 758–768, Seattle, Washington, USA, 10 2013. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/D13-1071.

Zhou, Hao, Yue Zhang, Shujian Huang, and Jiajun Chen. A Neural Probabilistic Structured-Prediction Model for Transition-Based Dependency 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 1: Long Papers)*, pages 1213–1222, Beijing, China, 7 2015. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P15-1117.

Appendix A. Deriving a dynamic oracle

In this appendix, we describe more precisely the dynamic oracle framework and how to derive such oracles, as well as some alterations made to the original framework to extend its applicability. Indeed, PanParser does not implement the action cost, but rather a generalized version which simplifies the oracle derivation for some systems, and also enables straightforward approximations in cases when dynamic oracles have only been inexactly derived (as done by Aufrant et al. (2018)) or not derived at all. We discuss here the formal grounding of this extension, as well as its limits.

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 computing the maximum UAS achieved on the given example, or (without normalization) the maximal number of correct attachments (henceforth denoted CA) on that sentence. So, the cost is evaluated by computing the maximum CA over all derivations resulting from the given configuration (c), and the maximum over only those following the given action (t).

$$\textsc{Cost}(c,t) = \left[\max_{t_1,\cdots,t_{end}} \mathrm{CA}(c \circ t_1 \circ \cdots \circ t_{end})\right] - \left[\max_{t_2,\cdots,t_{end}} \mathrm{CA}(c \circ t \circ t_2 \circ \cdots \circ t_{end})\right]$$

By definition of the maximum, Cost is always non-negative, 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 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 RelaxedCost function computes an overestimate of 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 RelaxedCost, but the actual cost can be retrieved by shifting all costs by the minimum RelaxedCost, which corresponds to the current 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, ExpectedCost is null, so RelaxedCost = Cost. Second, since $\min_t \textsc{Cost}(c,t) = 0$, in both cases (RelaxedCost and Cost), shifting by the minimum cost always yields Cost values.

Consequently, in PanParser, we define optimal actions as *minimum-cost actions*, and transition system implementations are supposed to compute either one of Cost (when computed as a loss difference) and 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.

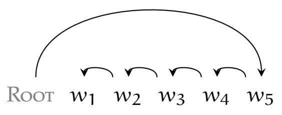
So, what happens if we have missed a given type of incompatibility? Or worse, if we miss all of them and simply use ForbiddenArcs for a non-arc-decomposable system? Using minimum-cost instead of zero-cost actions in fact strongly alleviates such issues.

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.

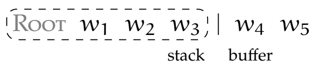
The only consequence on training is that the cost under-estimation introduces for the oracle a preference towards late resolution of inconsistencies: in case of two incompatible arcs, the parser will prefer actions that keep both options as long as possible over actions that forbid one of them right away. Figure 2 shows how bad this tendency can be. However, whether such cases have a strong impact on accuracy remains to be ascertained, on a per-case basis.

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. We thus advocate to empirically assess its effects for each considered system.

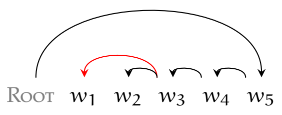
Non-projective examples Aufrant et al. (2018) have shown how dynamic oracles (with a minimum-cost criterion) make it possible to train a projective parser on non-projective sentences; this directly results from their ability to *accept* past errors and do



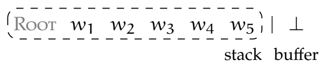
(a) Reference parse tree.



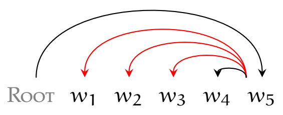
(b) Stack and buffer of the (already suboptimal) configuration to evaluate; all stack elements are unattached.



(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 2: Consequences of training with an under-estimated cost ignoring arc incompatibilities, using ArcStandard with Root in first position.

their best to select good decisions anyway. 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 is the case 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 or projectivizing all non-projective sentences (Aufrant et al., 2018).

Systems without known dynamic oracle Defining the action 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 designated by the heuristics used for pre-computing references is given cost 0, all other transitions are given cost 1. This method ensures that any transition system can be incorporated in the PanParser, even though in this case it will not be fully compatible with other components (no dynamic oracle, so no error exploration and no constrained parsing).

Appendix B. Global dynamic oracles

In PanParser, the training procedure is mostly 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 3 (on the following page) shows how Correct is used to apply the early-update and max-violation strategies with dynamic oracles.

Algorithm 3: Global dynamic oracle: error criterion and choice of an update configuration pair.

```
$c_0$: configuration to start decoding from
$\mathrm{top}_\theta(\cdot)$: best scoring element according to $\theta$
$\textsc{Next}(c)$: the set of all successors of $c$ (or only $c$ if it is final)
Function FindViolation($c_0$, $y$, $\theta$)
    Beam $\leftarrow \{c_0\}$
    while $\exists c \in Beam, \neg\textsc{Final}(c)$ do
        Succ $\leftarrow \cup_{c \in \mathrm{Beam}} \textsc{Next}(c)$
        Beam $\leftarrow$ k-best(Succ, $\theta$)
        if $\forall c \in Beam, \neg\textsc{Correct}_y(c|c_0)$ then
            gold $\leftarrow \{c \in \mathrm{Succ}|\textsc{Correct}_y(c|c_0)\}$
            return gold, Beam
    gold $\leftarrow \{c \in \mathrm{Beam}|\textsc{Correct}_y(c|c_0)\}$
    return gold, Beam
Function EarlyUpdateOracle($c_0$, $y$, $\theta$)
    gold, Beam $\leftarrow \textsc{FindViolation}(c_0, y, \theta)$
    return $\mathrm{top}_\theta$(gold), $\mathrm{top}_\theta$(Beam)
Function MaxViolationOracle($c_0$, $y$, $\theta$)
    gold, Beam $\leftarrow \textsc{FindViolation}(c_0, y, \theta)$
    candidates $\leftarrow \{(\mathrm{top}_\theta(\mathrm{gold}), \mathrm{top}_\theta(\mathrm{Beam}))\}$
    while $\exists c \in Beam, \neg\textsc{Final}(c)$ do
        Succ $\leftarrow \cup_{c \in \mathrm{Beam}} \textsc{Next}(c)$
        Beam $\leftarrow$ k-best(Succ, $\theta$)
        $\mathrm{Succ}^+ \leftarrow \cup_{c \in \mathrm{gold}} \{c' \in \textsc{Next}(c)|\textsc{Correct}_y(c'|c_0)\}$
        gold $\leftarrow$ k-best($\mathrm{Succ}^+$, $\theta$)
        candidates $\leftarrow$ candidates $+ (\mathrm{top}_\theta(\mathrm{gold}), \mathrm{top}_\theta(\mathrm{Beam}))$
    return $\arg\max_{c^+, c^- \in \mathrm{candidates}} (\mathrm{score}_\theta\,(c^-) - \mathrm{score}_\theta\,(c^+))$
```

Address for correspondence:
Lauriane Aufrant
lauriane.aufrant@limsi.fr
LIMSI – 508 rue John von Neumann, 91405 Orsay, France