Right-branching Tree Transformation for Eager Dependency Parsing

Xiaoye Wu and **David A. Smith**
Center for Intelligent Information Retrieval
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, U.S.A.
{xiaoye,dasmith}@cs.umass.edu

Abstract

This paper presents a reversible transformation on dependency trees that produces purely right-branching structures. On their own, these structures are easier for stack-based shift-reduce parsers to learn. We describe experiments training parsers on the transformed trees, parsing raw text, and then reversing the transformation on the results to evaluate. While these conditions do not significantly change parsing accuracy, a parser tailored to exploit the greater predictability of the transformed trees is able to run ten times faster than the widely-used MaltParser system, while using the same modeling and learning infrastructure. We conclude with experiments that explore further search strategies enabled by the transformed structures.

1 Introduction

Current research in data-driven dependency parsing can be separated into *graph-based* and *transition-based* methods (Nivre and McDonald, 2008). Graph-based methods view dependency parsing a sentence as a structured prediction problem whose output is a single (labeled) directed graph. The features of this structured prediction model encode constraints about which edges, pairs, of edges, etc., should appear in this graph. If these constraints apply only to single edges, $O(n^3)$ algorithms optimally solve this graph-prediction problem for both projective and non-projective trees. Higher-order constraints can improve empirical accuracy and linguistic plausibility—at the cost of making optimal projective parsing slower and non-projective parsing intractable (McDonald and Satta, 2007). Many state-of-the-art graph-based methods thus turn to approximate inference techniques (McDonald and Pereira, 2006; Smith and Eisner, 2008; Martins et al., 2009).

Transition-based dependency parsers, in contrast, aim to select the a sequence of appropriate actions to take during a tree construction process (Nivre, 2008). This tree-construction often proceeds from the beginning to the end of the input sentence (incrementally), which more plausibly models human sentence processing. Transition-based parsers have the additional advantage that parsing time for projective trees is linearly dependent on the length of sentence (quadratic for non-projective). Among transition based parsers, stack-based shift-reduce parsers have shown state-of-art performance by making shift and attachment decisions based on local features.

There is, however, a fundamental asymmetry in these stack-based incremental algorithms. When a token is a left branch child of its parent, meaning it appears before its dependency parent in the sentence, the parser does not have any knowledge of its parent and must make a shift decision based on the absence of certain features on the stack. On the other hand, when the parent precedes the child token, the parser makes an attachment decision based on the presence of certain features. In a sense, the shift decision on a left branching child delays its attachment until some expected features show up. Successful

shift-reduce parsers often use look-ahead methods, such as reading more tokens from the input buffer, to get around this asymmetry. Nevertheless, the number of tokens to be considered is often arbitrary and perhaps inadequate to deal with longer span left arcs.

In this paper, we present a right-branching transformation that aims to eliminate this asymmetry. We describe the transformation algorithm in section 2. In section 3, we present an experiment that studies the effect of the transformation on an existing shift-reduce parser. In section 4, we describe a new parser that is tailored to deal with right-branching trees specifically. In section 5, we analyse the potential benefit of the transformation in a more global parsing strategy, such as beam search and backtracking.

2 A right-branching transformation for dependency trees

The asymmetry between left and right children in a dependency tree often requires a shift-reduce parser to delay an attachment decision until all relevant tokens are visible to the parser. We can eliminate the need to delay decisions by transforming all dependency trees into right branching trees. In right branching trees, all children must appear after the parent token in the sentence. Therefore each token must always make an attachment when it shows up on top of the input buffer, without delay.

If we apply such a transformation, we want to preserve all the structural information of the original tree, which is equivalent to saying that there must exist a reverse transformation that will recover the original tree. To make the transformation reversible, we introduce two types of marking on the arcs that need to be changed (figure 1). The first type is a "reversed" arc, which is the result of reversing the first left-branched arc spanned by any head token. After this reversal, the arc originally pointing into the head token will point into the child token. There possibly exist other left children between the first left child and the head. Arcs pointing into these tokens originally come from the head; after the reversion, we create new arcs from the first left

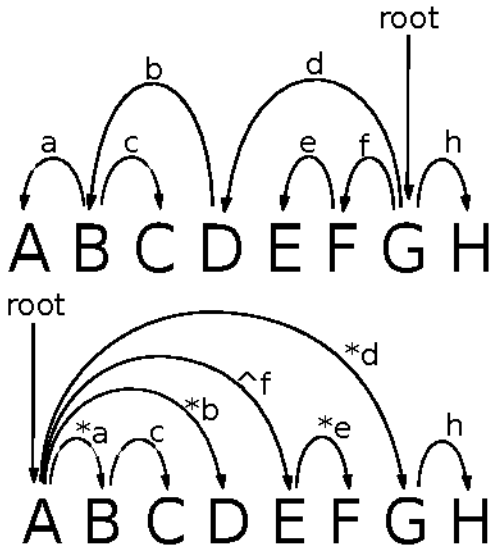

Figure 1: A dependency tree before and after transformation. Lower-case letters correspond to link targets in the original tree to clarify correspondences. Asterisks (*) mark reversed arcs, and carets (^) mark relocated arcs.

child to these tokens and mark them as "relocated" arcs. As a result of the transformation, a head token with n left children will become a structure whose head is the first token and have $n-1$ relocated arcs followed by one reversed arc. If a token is a descendant nested in a series of left branches, after the transformation it will have several outgoing reversed arcs. Note that the transformation does not make any changes on right-branching arcs.

Figures 2 and 3 show the transformation and reverse-transformation algorithms in pseudo-code.

2.1 Proof of reversibility

To prove that each dependency tree has a unique transformation, we only need to show that the transformation is a one-to-one mapping for any subtree consisting of a head and its children. The argument for the whole tree follows immediately as the proof can be applied recursively.

Let a, b be two subtrees. Since the transformation does not change the tokens, their basic dependency relationships or their orders, if $a_i \neq b_i$ or $deprel(a_i) \neq deprel(b_i)$ for any index i, the

```
function transform(dependency tree s)
  for every token t in s:
    while head(t) appears after t:
      for every token c between t and head(t):
        if head(c) equals head(t):
          mark deprel(c) as relocated
          set head(c) to t
      exchange deprel(t) with deprel_of(head(t))
      mark deprel(head(t)) as reversed
      let h = head(t)
      set head(t) to head(head(t))
      set head(h) to t
```

Figure 2: Algorithm to transform to original to right-branching trees.

```
function reverse(dependency tree s)
  for every token t in s in reverse order:
    while deprel(t) is marked as reversed:
      for every token c between t and head(t):
        if head(c) equals head(t)
        and deprel(c) is relocated:
          unmark deprel(c) as relocated
          set head(c) to t
      exchange deprel(t) with deprel(head(t))
      unmark deprel(head(t)) as reversed
      let h = head(t)
      set head(t) to head(head(t))
      set head(h) to t
```

Figure 3: Algorithm to reverse right-branching transformation.

result of their transformations $F(a)$, $F(b)$ will obviously be different. Let $head(a)$ denote the head token of a, then the transformation guarantees that: 1. $head(a)$ is the only token whose dependency relationship is marked as reversed in $F(a)$; 2. any token a_j where $j <$ index of $head(a)$ is a child of a_0 with its dependency relationship marked as relocated in $F(a)$; 3. any token a_k where $j >$ index of $head(a)$ remains a child of $head(a)$ with its dependency relationship unchanged in $F(a)$. It is straight-forward to see that if $a \neq b$ in any way, $F(a)$ cannot be the same as $F(b)$. The proof for the reverse transformation is similar.

3 Experiment 1: Learnability of transformed trees

A parser that deals with right branching trees needs only two kinds of actions: right-arc and reduce. The transformation in effect incorporates lookahead into the parsing algorithm. For example, the original parser must wait until a noun shows up after a determiner to fully link up the attachments. When parsing right-branching trees, in contrast, an eager parser will immediately attach the determiner to the stack and wait for a noun to attach to the determiner with a reversed arc. The set of dependency arcs a token accepts are determined by the token itself and all potential head tokens that can take the token as a left descendant, and making an arc is the same as expecting a specific type of head.

3.1 Experimental setup

Empirically, we have found that this transformation improves the predictability of arcs. The experiment set-up is as follows: we take section 2 to 22 of the Penn treebank converted into a dependency treebank (Johansson and Nugues, 2007) as training data, and section 23 as testing data. We first train a standard MaltParser (Nivre, 2008) and test it on the test set. We then train the same configuration of MaltParser on the transformed training set and test the parser on the transformed test set. Finally, we attempt to reverse the transformation on the output of the second parser and test it on the original test set. The

training/test	reg./reg.	xfm./xfm.	rev./reg.
labeled attachment	78.98	80.97	77.63
unlabeled attachment	82.79	87.87	81.72
label accuracy	83.93	84.62	83.87

Table 1: Percentage accuracy scores of different training-test set-ups. reg. = regular, xfm. = transformed, rev. = reverse-transformed

results are summarized in table 1.

As can be seen, head, label and combined accuracy all went up for transformed parser on transformed test data. Unlabeled attachment score is most visibly enhanced, probably due to the reduction of attachment structural variations. The majority of phrases are now depth-one right-branching subtrees, and attaching to the first token of the phrase is an easy decision to make. Recognition of the internal structure now relies on the correct label on the dependency arc. For example, the noun phrase "the stock market" has an ambiguous attachment for the determiner. In a regular tree, the parser must learn not to hastily attach the determiner to "stock" if there is another noun after it. In the transformed tree, the parser always attach "stock" to "the", but it must make the correct decision that this attachment is relocated, rather than reversed.

There is a further migration of ambiguity observed in the results. For example, the head attachment of determiners is among the top frequent errors in both type of tree parses, but for different reasons. In regular trees, a noun phrase that involves a long sequence of NMODs creates a problem for determiner attachment. On the other hand, the error in transformed trees is accompanied by visible improvements of head attachment of nouns. This reflects the fact that the first token of a phrase, such as a determiner, is the head of a phrase in the transformed trees. The difficulty of predicting the parent token of a noun phrase now rests on the determiner instead of the noun in the phrase. A similar effect is seen in the enhancement of head attachment accuracy of verbs, whose preceding subjects now bear the burden of ambiguity.

The results above are evidence that the transformation of trees improves head and label predictability. We have also proved that the transformation is a bijection, so parsing transformed sentences is in theory as difficult as parsing original sentences. One can imagine many *non*-reversible transformation that would improve predictability: trees that attached each word to its predecessor would be extremely easy to learn! But the one-to-one relationship between original and transformed trees in the current setup precludes such degenerate cases.

When the transformation is reversed, however, the improvements are lost; therefore the reverse transformation must amplify the errors made in the attachments. There are two ways the reverse transformation amplify errors. The first is when a reversed attachment is erroneously made, every relocated attachment right before the reversed attachment will also be attached to the wrong head in this case. The second type of degradation comes from improper dependency sequences. In a properly transformed tree, any relocated or reversed attachment to a head token must occur after regular attachments, and any relocated attachment cannot be the last attachment of the head token. Failure to adhere to these rules will leave vestigial relocated attachments in the reversed tree and possibly generate ill-formed non-projective structures. The MaltParser obviously cannot ensure a proper attachment sequence, as it only makes highly local decisions.

4 Experiment 2: A parser for right-branching dependency trees

The added constraints of the transformed trees prompt us to modify the existing shift-reduce parser by training separate prediction models for when the head token previously emitted a regular, relocated, or reversed arc. This modification is based on the fact that the type of arcs a token

emits is conditioned on the previous arc it has previously emitted. When a token emits a relocated arc, it must next emit another relocated arc or a reversed arc, and it cannot be reduced. When a token emits a reversed arc, it cannot emit a regular arc afterwards, and it is much more likely to be reduced off the stack. We can treat these different "distributions" as the normal, relocated, and reversed state of the stack and train a classification model for each of them. These modifications aim to improve the parser performance in two ways: first, represent the decision "distribution" more accurately by imposing additional state conditions. Secondly, the constraints on the actions the parser can take will hopefully reduce ill-formatted trees that cannot be correctly reverse-transformed.

We built a new right-branching (RB) parser that uses three different prediction models for three states of the stack. Although we did not explicitly implement a backtracking mechanism, the MaltParser style "look ahead" features provide limited effect of a backtracking system. In theory a parser that parses right branching trees only needs to extract features from the stack and the first token on the input buffer, but look-ahead features allow the parser to choose a state more carefully.

There exist some correspondences between features of MaltParser and features of our new parser, but many of the MaltParser features are no longer needed. All features extracted from left children are no longer useful, as there are no left children. None of the tokens in the input buffer have any dependency arcs attached, and functions like "the head of a token" are translated to "the previous token on stack". A crucial difference in features is that while MaltParser uses features such as "the POS tag of i-th token on the stack", our parser uses "the true POS tag of the i-th token on the stack". The "true token" of a token is defined as follows: if a token has never emitted a reversed arc, its true token is itself; otherwise its true head is its last child whose attachment label is reversed. The notion of true head ensures that arcs are decided on relevant information. For example, if the first three tokens of the sentence "the stock market crashed" are parsed, after appropriate reductions, "the" is the only token left there as the head of the noun phrase. The decision to attach "crashed" to the stack token "the" is essentially the decision to attach the head of the noun phrase, "market", to the verb "crashed" as its subject. Here the true token of "the" is appropriately the syntactical head of the noun phrase, "market", which replaces the features of "the" to help the parser to make the correct attachment. Another modified feature is ranged look ahead features, which signal the existence of certain features in some ranges of the input buffer. The ranged feature that looks for all POS tags from the second to the forth token on the input buffer is used in the right-branching parser.

For the RB parser, we used a feature set that roughly corresponds to the one used by the default MaltParser, as listed in table 2. Since all the left-child features originally employed by MaltParser are no longer useful, we allowed the parser to extract the POS tag of one more token in the stack. This is justified by the fact that a left child of a stack token in MaltParser will end up as a token deeper in the stack in the RB parser, emitting a reversed arc.

The performance of the RB parser closely matches that of MaltParser (table 3). The labeled attachment score is 78.53, insignificantly lower than MaltParser. Unlabeled attachment score becomes 82.42; label accuracy becomes 84.68, higher than MaltParser's 83.93. As expected, we nearly eliminated the vestigial arcs in the parser output compared to that of a MaltParser trained with transformed trees (14 versus 224). The labeled attachment, unlabeled attachment and label scores of transformed output over transformed test set are 81.51, 87.90 and 85.35, respectively. The same improvements of head and label predictability are observed. Ambiguity (e.g. between determiners and nouns) also shifted similarly to the results of experiment 1. A final point of interest is that the RB parser achieves higher accuracies on smaller amounts of training data.

Although on the full training set the transfor-

MaltParser	RB Parser
form(input[0]) form(input[1]) form(stack[0]) form(head(stack[0]))	form(input[0]) form(input[1]) form(stack[0]) form(stack[1])
POStag(input[0]) POStag(input[1]) POStag(input[2]) POStag(input[3]) POStag(stack[0]) POStag(stack[1])	POStag(input[0]) POStag(input[1]) POStag(input[2]) POStag(input[3]) POStag(stack[0]) POStag(stack[1]) POStag(stack[3])
deprel(stack[0]) deprel(ldep(stack[0])) deprel(rdep(stack[0])) deprel(ldep(input[0]))	deprel(stack[0]) deprel(rdep(stack[0]))

Table 2: Corresponding features employed by MaltParser and Right-Branching Parser

mation does not significantly improve parsing accuracy, it has a large advantage in parsing time. On a 2.1GHz processor, we ran the RB parser and MaltParser on all 24 sections of the Penn Treebank WSJ corpus. The RB parser spent 75.5 seconds to read and parse all 49,208 sentences, of which just under 64 seconds are spent on parsing. The transformation and reversal of all sentences took 24 seconds each. The sum of these is still smaller than the MaltParser by a factor of ten, which parsed the whole set in 996 seconds. Although both the MaltParser and the RB parser use the logistic regression learner in LIBLINEAR (Fan et al., 2008), the light weight of our RB parser implementation may have contributed to this speed-up. More importantly, the transformation on average lowers the number of possible actions given each parsing state, since left-attach actions are no longer used. Upon close examination, we discovered the set of possible actions is particularly small when the stack state is "relocated" or "reversed".

5 Experiment 3: Oracle experiment on the RB parser for a potential backtracking algorithm

The evaluations so far have shown that the transformation and reversal does not hurt the performance of a greedy (eager) shift-reduce parser based on local decisions. If we allow a slightly more global decision approach, for example beam search or backtracking, the transformation has the potential to greatly improve the accuracy and learnability of the parser. Particle filtering, which can be thought of as a stochastic variant of beam search, has also been shown to enhance parser performance (Levy et al., 2009).

Backtracking works as a depth-first analogue: the parser hypothesizes an attachment and proceed until it discovers the attachment causes difficulties in later parts of the sentence. We believe that the right-branching transformation can further improve parsing performance when combined with these techniques. There are two reasons this is likely to be the case. The first reason is that many attachment errors are amplified in the reverse transformation. The amplified errors often result in vestigial arcs and non-projective structures after reversal, which are found in the reversed output of both MaltParser (trained on right-branching trees) and RB parser. These facts lead us to believe that if we can reduce the amplification of errors introduced in the reversal process, we will improve the performance. More global algorithms, such as beam search and backtracking, will force the parsed trees

training size	1 sec. xfm.	1 sec. rev.	4 sec. xfm.	4 sec. rev.	20 sec. xfm.	20 sec. rev.
MaltParser	N/A	67 (73,74)	N/A	73 (78,79)	N/A	79 (83,84)
RB Parser	74 (84,79)	71 (77,78)	77 (86,82)	74 (80,80)	82 (88,85)	79 (82,85)
Oracle Parser	78 (88,81)	75 (82,81)	81 (89,84)	77 (83,83)	84 (90,87)	81 (85,86)

Table 3: Comparison between MaltParser, RB parser, and the oracle parser that knows when to backtrack. The evaluation shows labeled attachment scores, unlabeled attachment scores, and label accuracies, in that order. The RB parser can learn more quickly from smaller amounts of training data.

to respect the constraints of transformed trees and eliminate structures that are not properly reversible. For example, when the parser runs into the situation that no appropriate reversed arc can be made after a series of relocated arcs, it should backtrack to where the first relocated arc was made. If we ensure the attachments are proper, we will preserve more of the enhancement from increased attachment predictability.

The second reason is best illustrated with an oracle experiment. Suppose we have a backtracking parser that parses right-branching trees. When the top token of the input buffer is attached to and therefore moved to the stack, the correct label of the attachment is actually the label on the arc accepted by its true token. For example, in the sentence "Eve ate the apple", "the" attaches to "ate" as the OBJ. This decision reflects that the parser expects a noun to head the determiner. In a backtracking system, a parser may hypothesize a few possible heads for each token it moves onto the stack, and eventually rejects all but one hypothesis. This is equivalent to treating each token as several hypothesized true tokens. Our oracle experiment is set up to explore the effect of this backtracking. As an ideal case, we assume the system can always reject the incorrect hypothesis, so when a token is to be moved onto the stack (and thereafter), we allow the feature extractor to extract its true token in the gold tree (which may be farther down the input buffer than the parser can see). This simulates picking a hypothesis (the correct one) the backtracking system makes.

The comparison between this parser and MaltParser trained with one and four sections of treebank are summarized in table 3. The result is significantly better than the standard MaltParser and the RB parser. The output of the oracle parser still contains a few vestigial arcs. Ill-formatted structures are formed when the parser model predicts a reversed arc but reaches the end of the sentence, so the reversed arc is never realized. Were true backtracking implemented, we can further improve performance by getting rid of these types of mistakes. The parser also sees enhancements in both precision and recall of binned head direction (table 4) and longer distance head attachments (table 5).

6 Conclusion

The right-branching transformation is a one-to-one mapping from an ordinary labeled syntactic dependency tree to a right-branching tree. The transformation eliminates the asymmetry of information between left and right children at the cost of a larger set of labels. The transformed trees exhibit an increase in attachment head and label predictability, although this improvement is not fully carried over once the transformed parser output is reversed. A shift-reduce parser specifically tailored to deal with right-branching trees only needs to have right-attach and reduction actions, which results on average in smaller decision spaces and therefore faster parsing. Our implementation of the parser is a factor of ten faster than the widely used shift-reduce MaltParser, while matching its performance.

The right-branching transformation also clearly has potential to help build a high performance shift-reduce dependency parser. Its improvement on predictability of attachment head and labels can be carried over if the parser implements a more global decision strategy such

direction	recall		precision	
	Malt	OP	Malt	OP
to_root	73.93	88.13	80.17	87.08
left	94.24	96.09	92.93	95.86
right	92.00	94.86	92.80	95.27

Table 4: Precision and recall of binned head direction, a comparison between MaltParser and the Oracle Parser (OP), trained on 20 sections of the Penn Treebank.

distance	recall		precision	
	Malt	OP	Malt	OP
1	93.53	91.08	94.06	94.81
2	86.39	88.30	85.47	88.18
3-6	76.08	78.95	80.18	80.50
7+	63.66	75.47	51.25	53.25

Table 5: Precision and recall of binned head distance, a comparison between MaltParser and the Oracle Parser (OP), trained on 20 sections of the Penn Treebank.

as beam search or backtracking.

7 Acknowledgement

This work was supported in part by the Center for Intelligent Information Retrieval, in part by NSF IIS-0910884, and in part by NEH grant #HD-50794-09. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

References

Fan, R.-E., K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. 2008. Liblinear: A library for large linear classification. *JMLR*, 9:1871–1874.

Johansson, Richard and Pierre Nugues. 2007. Extended constituent-to-dependency conversion for English. In *NODALIDA*.

Levy, Roger, Florencia Reali, and Thomas L. Griffiths. 2009. Modeling the effects of memory on human online sentence processing with particle filters. In *NIPS*.

Martins, Andre, Noah Smith, and Eric Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *ACL*, pages 342–350.

McDonald, Ryan and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *EACL*.

McDonald, Ryan and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *IWPT*, pages 121–132.

Nivre, Joakim and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *ACL*, pages 950–958.

Nivre, Joakim. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Smith, David A. and Jason Eisner. 2008. Dependency parsing by belief propagation. In *EMNLP*, pages 145–156.