# A Non-Monotonic Arc-Eager Transition System for Dependency Parsing

**Matthew Honnibal**
Department of Computing
Macquarie University
Sydney, Australia
matthew.honnibal@mq.edu.edu.au

**Yoav Goldberg**
Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
yoav.goldberg@gmail.com

**Mark Johnson**
Department of Computing
Macquarie University
Sydney, Australia
mark.johnson@mq.edu.edu.au

## Abstract

Previous incremental parsers have used monotonic state transitions. However, transitions can be made to revise previous decisions quite naturally, based on further information.

We show that a simple adjustment to the Arc-Eager transition system to relax its monotonicity constraints can improve accuracy, so long as the training data includes examples of mistakes for the non-monotonic transitions to repair. We evaluate the change in the context of a state-of-the-art system, and obtain a statistically significant improvement ($p < 0.001$) on the English evaluation and 5/10 of the CoNLL languages.

## 1 Introduction

Historically, monotonicity has played an important role in transition-based parsing systems. Non-monotonic systems, including the one presented here, typically redundantly generate multiple derivations for each syntactic analysis, leading to *spurious ambiguity* (Steedman, 2000). Early, pre-statistical work on transition-based parsing such as Abney and Johnson (1991) implicitly assumed that the parser searches the entire space of possible derivations. The presence of spurious ambiguity causes this search space to be a directed graph rather than a tree, which considerably complicates the search, so spurious ambiguity was avoided whenever possible.

However, we claim that non-monotonicity and spurious ambiguity are not disadvantages in a modern statistical parsing system such as ours. Modern statistical models have much larger search spaces because almost all possible analyses are allowed, and a numerical score (say, a probability distribution) is used to distinguish better analyses from worse ones. These search spaces are so large that we cannot exhaustively search them, so instead we use the scores associated with partial analyses to guide a search that explores only a minuscule fraction of the space (In our case we use greedy decoding, but even a beam search only explores a small fraction of the exponentially-many possible analyses).

In fact, as we show here the additional redundant pathways between search states that non-monotonicity generates can be advantageous because they allow the parser to "correct" an earlier parsing move and provide an opportunity to recover from formerly "fatal" mistakes. Informally, non-monotonicity provides "many paths up the mountain" in the hope of making it easier to find at least one.

We demonstrate this by modifying the Arc-Eager transition system (Nivre, 2003; Nivre et al., 2004) to allow a limited capability for non-monotonic transitions. The system normally employs two deterministic constraints that limit the parser to actions consistent with the previous history. We remove these, and update the transitions so that conflicts are resolved in favour of the latest prediction.

The non-monotonic behaviour provides an improvement of up to 0.2% accuracy over the current state-of-the-art in greedy parsing. It is possible to implement the greedy parser we describe very efficiently: our implementation, which can be found at `http://www.github.com/syllog1sm/redshift`, parses over 500 sentences a second on commodity hardware.



*Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172,
Sofia, Bulgaria, August 8-9 2013. 

## 2 The Arc-Eager Transition System

In transition-based parsing, a parser consists of a state (or a configuration) which is manipulated by a set of actions. An action is applied to a state and results in a new state. The parsing process concludes when the parser reaches a final state, at which the parse tree is read from the state. A particular set of states and actions yield a transition-system. Our starting point in this paper is the popular Arc-Eager transition system, described in detail by Nivre (2008).

The state of the arc-eager system is composed of a stack, a buffer and a set of arcs. The stack and the buffer hold the words of a sentence, and the set of arcs represent derived dependency relations.

We use a notation in which the stack items are indicated by $S_i$, with $S_0$ being the top of the stack, $S_1$ the item previous to it and so on. Similarly, buffer items are indicated as $B_i$, with $B_0$ being the first item on the buffer. The arcs are of the form $(h, l, m)$, indicating a dependency in which the word $m$ modifies the word $h$ with label $l$.

In the initial configuration the stack is empty, and the buffer contains the words of the sentence followed by an artificial ROOT token, as suggested by Ballesteros and Nivre (2013). In the final configuration the buffer is empty and the stack contains the ROOT token.

There are four parsing actions (Shift, Left-Arc, Right-Arc and Reduce, abbreviated as S,L,R,D respectively) that manipulate stack and buffer items. The **Shift** action pops the first item from the buffer and pushes it on the stack (the Shift action has a natural precondition that the buffer is not empty, as well as a precondition that ROOT can only be pushed to an empty stack). The **Right-Arc** action is similar to the Shift action, but it also adds a dependency arc $(S_0, B_0)$, with the current top of the stack as the head of the newly pushed item (the Right action has an additional precondition that the stack is not empty).[1] The **Left-Arc** action adds a dependency arc $(B_0, S_0)$ with the first item in the buffer as the head of the top of the stack, and pops the stack (with a precondition that the stack and buffer are not empty, and that $S_0$ is not assigned a head yet). Finally, the **Reduce** action pops the stack, with a precondition that the stack is not empty and that $S_0$ is already assigned a head.

[1]For labelled dependency parsing, the Right-Arc and Left-Arc actions are parameterized by a label $L$ such that the action $\text{Right}_L$ adds an arc $(S_0, L, B_0)$, similarly for $\text{Left}_L$.

### 2.1 Monotonicty

The preconditions of the Left-Arc and Reduce actions ensure that every word is assigned exactly one head, resulting in a well-formed parse tree. The single head constraint is enforced by ensuring that once an action has been performed, subsequent actions must be consistent with it. We refer to this consistency as the *monotonicity* of the system.

Due to monotonicity, there is a natural pairing between the Right-Arc and Reduce actions and the Shift and Left-Arc actions: a word which is pushed into the stack by Right-Arc must be popped using Reduce, and a word which is pushed by Shift action must be popped using Left-Arc. As a consequence of this pairing, a Right-Arc move determines that the head of the pushed token must be to its left, while a Shift moves determines a head to its right. Crucially, the decision whether to Right-Arc or Shift is often taken in a state of missing information regarding the continuation of the sentence, forcing an incorrect head assignment on a subsequent move.

Consider a sentence pair such as (a)"I saw Jack and Jill" / (b)"I saw Jack and Jill fall". In (a), "Jack and Jill" is the NP object of "saw", while in (b) it is a subject of the embedded verb "fall". The monotonic arc-eager parser has to decide on an analysis as soon as it sees "saw" on the top of the stack and "Jack" at the front of the buffer, without access to the disambiguating verb "fall".

In what follows, we suggest a non-monotonic variant of the Arc-Eager transition system, allowing the parser to recover from the incorrect head assignments which are forced by an incorrect resolution of a Shift/Right-Arc ambiguity.

## 3 The Non-Monotonic Arc-Eager System

The Arc-Eager transition system (Nivre et al., 2004) has four moves. Two of them create dependencies, two push a word from the buffer to the stack, and two remove an item from the stack:

| | Push | Pop |
|---|---|---|
| Adds dependency | *Right-Arc* | **Left-Arc** |
| No new dependency | **Shift** | *Reduce* |

Every word in the sentence is pushed once and popped once; and every word must have exactly one head. This creates two pairings, along the diagonals: (S, L) and (R, D). Either the push move adds the head or the pop move does, but not both and not neither.

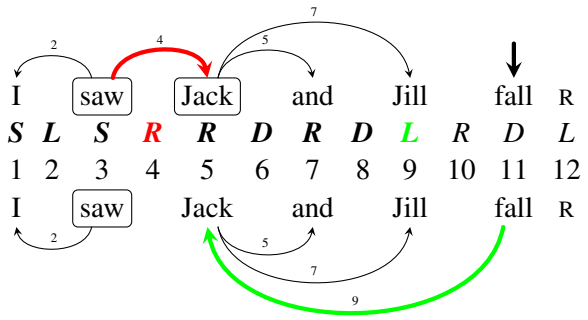


Figure 1: State before and after a non-monotonic Left-Arc. At move 9, *fall* is the first word of the buffer (marked with an arrow), and *saw* and *Jack* are on the stack (circled). The arc created at move 4 was incorrect (in red). Arcs are labelled with the move that created them. After move 9 (the lower state), the non-monotonic Left-Arc move has replaced the incorrect dependency with a correct Left-Arc (in green).

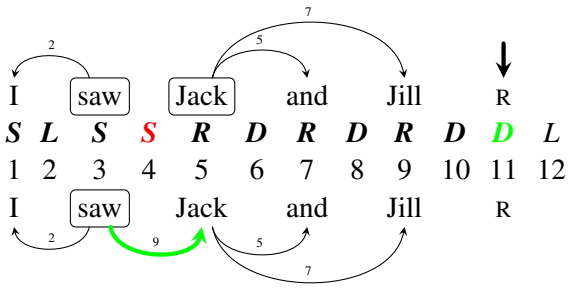


Figure 2: State before and after a non-monotonic Reduce. After making the wrong push move at 4, at move 11 the parser has *Jack* on the stack (circled), with only the dummy ROOT token left in the buffer. A monotonic parser must deterministically Left-Arc *Jack* here to preserve the previous decision, despite the current state. We remove this constraint, and instead assume that when the model selects Reduce for a headless item, it is reversing the previous Shift/Right decision. We add the appropriate arc, assigning the label that scored highest when the Shift/Right decision was made.

Thus in the Arc-Eager system the first move determines the corresponding second move. In our non-monotonic system the second move can overwrite an attachment made by the first. This change makes the transition system *non-monotonic*, because if the model decides on an incongruent pairing we will have to either undo or add a dependency, depending on whether we correct a prior Right-Arc, or a prior Shift.

### 3.1 Non-monotonic Left-Arc

Figure 1 shows a before-and-after view of a non-monotonic transition. The sequence below the words shows the transition history. The words that are circled in the upper and lower line are on the stack before and after the transition, respectively. The arrow shows the start of the buffer, and arcs are labelled with the move that added them.

The parser began correctly by Shifting *I* and Left-Arcing it to *saw*, which was then also Shifted. The mistake, made at Move 4, was to Right-Arc *Jack* instead of Shifting it.

The difficulty of this kind of a decision for an incremental parser is fundamental. The leftward context does not constrain the decision, and an arbitrary amount of text could separate *Jack* from *fall*. Eye-tracking experiments show that humans often perform a saccade while reading such examples (Frazier and Rayner, 1982).

In moves 5-8 the parser correctly builds the rest of the NP, and arrives at *fall*. The monotonicity constraints would force an incorrect analysis, having *fall* modify *Jack* or *saw*, or having *saw* modify *fall* as an embedded verb with no arguments.

We allow Left-Arcs to 'clobber' edges set by Right-Arcs if the model recommends it. The previous edge is deleted, and the Left-Arc proceeds as normal. The effect of this is exactly as if the model had correctly chosen Shift at move 4. We simply give the model a second chance to make the correct choice.

### 3.2 Non-monotonic Reduce

The upper arcs in Figure 2 show a state resulting from the opposite error. The parser has Shifted *Jack* instead of Right-Arcing it. After building the NP the buffer is exhausted, except for the ROOT token, which is used to wrongly Left-Arc *Jack* as the sentence's head word.

Instead of letting the previous choice lock us in to the pair (Shift, Left-Arc), we let the later decision reverse it to (Right-Arc, Reduce), if the parser has predicted Reduce in spite of the signal from its previous decision. In the context shown in Figure 2, the correctness of the Reduce move is quite predictable, once the choice is made available.

When the Shift/Right-Arc decision is reversed, we add an arc between the top of the stack ($S_0$) and the word preceding it ($S_1$). This is the arc that would have been created had the parser chosen to Right-Arc when it chose to Shift. Since our idea is to reverse this mistake, we select the Right-Arc label that the model scored most highly at that time.[2]

[2] An alternative approach to label assignment is to parameterize the Reduce action with a label, similar to the Right-Arc and Left-Arc actions, and let that label override the previously predicted label. This would allow the parser to con-

**To summarize,** our Non-Monotnonic Arc-Eager system differs from the monotonic Arc-Eager system by:

- Changing the Left-Arc action by removing the precondition that $S_0$ does not have a head, and updating the dependency arcs such previously derived arcs having $S_0$ as a dependent are removed from the arcs set.
- Changing the Reduce action by removing the precondition that $S_0$ has a head, and updating the dependency arcs such that if $S_0$ does not have a head, $S_1$ is assigned as the head of $S_0$.

## 4 Why have two push moves?

We have argued above that it is better to trust the second decision that the model makes, rather than using the first decision to determine the second. If this is the case, is the first decision entirely redundant? Instead of defining how pop moves can correct Shift/Right-Arc mistakes, we could instead eliminate the ambiguity. There are two possibilities: Shift every token, and create all Right-Arcs via Reduce; or Right-Arc every token, and replace them with Left-Arcs where necessary.

Preliminary experiments on the development data revealed a problem with these approaches. In many cases the decision whether to Shift or Right-Arc is quite clear, and its result provides useful conditioning context to later decisions. The information that determined those decisions is never lost, but saving all of the difficulty for later is not a very good structured prediction strategy.

As an example of the problem, if the Shift move is eliminated, about half of the Right-Arcs created will be spurious. All of these arcs will be assigned labels making important features uselessly noisy. In the other approach, we avoid creating spurious arcs, but the model does not predict whether $S_0$ is attached to $S_1$, or what the label would be, and we miss useful features.

The non-monotonic transition system we propose does not have these problems. The model learns to make Shift vs. Right-Arc decisions as normal, and conditions on them — but without *committing* to them.

dition its label decision on the new context, which was sufficiently surprising to change its move prediction. For efficiency and simplicity reasons, we chose instead to trust the label the model proposed when the reduced token was initially pushed into the stack. This requires an extra vector of labels to be stored during parsing.

## 5 Dynamic Oracles

An essential component when training a transition-based parser is an oracle which, given a gold-standard tree, dictates the sequence of moves a parser should make in order to derive it. Traditionally, these oracles are defined as functions from trees to sequences, mapping a gold tree to a single sequence of actions deriving it, even if more than one sequence of actions derives the gold tree. We call such oracles *static*. Recently, Goldberg and Nivre (2012) introduced the concept of a *dynamic* oracle, and presented a concrete oracle for the arc-eager system. Instead of mapping a gold tree to a sequence of actions, the dynamic oracle maps a $\langle$configuration, gold tree$\rangle$ pair to a *set* of optimal transitions. More concretely, the dynamic oracle presented in Goldberg and Nivre (2012) maps $\langle \text{action}, \text{configuration}, \text{tree} \rangle$ tuples to an integer, indicating the number of gold arcs in $tree$ that can be derived from $configuration$ by some sequence of actions, but could not be derived after applying $action$ to the configuration.

There are two advantages to this. First, the ability to label any configuration, rather than only those along a single path to the gold-standard derivation, allows much better training data to be generated. States come with realistic histories, including errors — a critical point for the current work. Second, the oracle accounts for spurious ambiguity correctly, as it will label multiple actions as correct if the optimal parses resulting from them are equally accurate.

In preliminary experiments in which we trained the parser using the static oracle but allowed the non-monotonic repair operations during parsing, we found that the the repair moves yielded no improvement. This is because the static oracle does not generate any examples of the repair moves during training, causing the parser to rarely predict them in test time. We will first describe the Arc-Eager dynamic oracle, and then define dynamic oracles for the non-monotonic transition systems we present.

### 5.1 Monotonic Arc-Eager Dynamic Oracle

We now briefly describe the dynamic oracle for the arc-eager system. For more details, see Goldberg and Nivre (2012). The oracle is computed by reasoning about the arcs which are reachable from a given state, and counting the number of gold arcs which will no longer be reachable after applying a

given transition at a given state. [3]

The reasoning is based on the observations that in the arc-eager system, new arcs $(h, l, m)$ can be derived iff the following conditions hold:
(a) There is no existing arc $(h', l', m)$ such that $h' \neq h$, and (b) Either both $h$ and $m$ are on the buffer, or one of them is on the buffer and the other is on the stack. In other words:
(a) once a word acquires a head (in a Left-Arc or Right-Arc transition) it loses the ability to acquire any other head.
(b) once a word is moved from the buffer to the stack (Shift or Right-Arc) it loses the ability to acquire heads that are currently on the stack, as well as dependents that are currently on the stack and are not yet assigned a head.[4]
(c) once a word is removed from the stack (Left-Arc or Reduce) it loses the ability to acquire any dependents on the buffer.
Based on these observations, Goldberg and Nivre (2012) present an oracle $\mathcal{C}(a, c, t)$ for the monotonic arc-eager system, computing the number of arcs in the gold tree $t$ that are reachable from a parser's configuration $c$ and are no longer reachable from the configuration $a(c)$ resulting from the application of action $a$ to configuration $c$.

### 5.2 Non-monotonic Dynamic Oracles

Given the oracle $\mathcal{C}(a, c, t)$ for the monotonic system, we adapt it to a non-monotonic variant by considering the changes from the monotonic to the non-monotonic system, and adding $\Delta$ terms accordingly. We define three novel oracles: $\mathcal{C}_{NML}$, $\mathcal{C}_{NMD}$ and $\mathcal{C}_{NML+D}$ for systems with a non-monotonic Left-Arc, Reduce or both.

$$
\begin{aligned}
\mathcal{C}_{NML}(a,c,t) &= \mathcal{C}(a,c,t) + \Delta_{NML}(a,c,t) \\
\mathcal{C}_{NMD}(a,c,t) &= \mathcal{C}(a,c,t) + \Delta_{NMD}(a,c,t) \\
\mathcal{C}_{NML+D}(a,c,t) &= \mathcal{C}(a,c,t) + \Delta_{NML}(a,c,t) \\
&\quad + \Delta_{NMD}(a,c,t)
\end{aligned}
$$

The terms $\Delta_{NML}$ and $\Delta_{NMD}$ reflect the score adjustments that need to be done to the arc-eager oracle due to the changes of the Left-Arc and Reduce actions, respectively, and are detailed below.

*Changes due to non-monotonic Left-Arc:*

- $\Delta_{NML}(\text{RIGHTARC}, c, t)$: The cost of Right-Arc is decreased by 1 if the gold head of $B_0$ is on the buffer (because $B_0$ can still acquire its correct head later with a Left-Arc action). It is increased by 1 for any word $w$ on the stack such that $B_0$ is the gold parent of $w$ and $w$ is assigned a head already (in the monotonic oracle, this cost was taken care of when the word was assigned an incorrect head. In the non-monotonic variant, this cost is delayed).
- $\Delta_{NML}(\text{REDUCE}, c, t)$: The cost of Reduce is increased by 1 if the gold head of $S_0$ is on the buffer, because removing $S_0$ from the stack precludes it from acquiring its correct head later on with a Left-Arc action. (This cost is paid for in the monotonic version when $S_0$ acquired its incorrect head).
- $\Delta_{NML}(\text{LEFTARC}, c, t)$: The cost of Left-Arc is increased by 1 if $S_0$ is already assigned to its gold parent. (This situation is blocked by a precondition in the monotonic case). The cost is also increased if $S_0$ is assigned to a non-gold parent, and the gold parent is in the buffer, but not $B_0$. (As a future non-monotonic Left-Arc is prevented from setting the correct head.)
- $\Delta_{NML}(\text{SHIFT}, c, gold)$: The cost of Shift is increased by 1 for any word $w$ on the stack such that $B_0$ is the gold parent of $w$ and $w$ is assigned a head already. (As in Right-Arc, in the monotonic oracle, this cost was taken care of when $w$ was assigned an incorrect head.)

*Changes due to non-monotonic Reduce:*

- $\Delta_{NMD}(\text{SHIFT}, c, gold)$: The cost of Shift is decreased by 1 if the gold head of $B_0$ is $S_0$ (Because this arc can be added later on with a non-monotonic Reduce action).
- $\Delta_{NMD}(\text{LEFTARC}, c, gold)$: The cost of Left-Arc is increased by 1 if $S_0$ is not assigned a head, and the gold head of $S_0$ is $S_1$ (Because this precludes adding the correct arc with a Reduce of $S_0$ later).
- $\Delta_{NMD}(\text{REDUCE}, c, gold) = 0$. While it may seem that a change to the cost of a Reduce action is required, in fact the costs of the monotonic system hold here, as the head of $S_0$ is

[3] The correctness of the oracle is based on a property of the arc-eager system, stating that if a set of arcs which can be extended to a projective tree can be individually derived from a given configuration, then a projective tree containing all of the arcs in the set is also derivable from the same configuration. This same property holds also for the non-monotonic variants we propose.

[4] The condition that the words on the stack are not yet assigned a head is missing from (Goldberg and Nivre, 2012)

predetermined to be $S_1$. The needed adjustments are taken care of in Left-Arc and Shift actions.[5]

- $\Delta_{NMD}(\text{RIGHTARC}, c, gold) = 0$

## 6 Applying the Oracles in Training

Once the dynamic-oracles for the non-monotonic system are defined, we could in principle just plug them in the perceptron-based training procedure described in Goldberg and Nivre (2012). However, a tacit assumption of the dynamic-oracles is that all paths to recovering a given arc are treated equally. This assumption may be sub-optimal for the purpose of training a parser for a non-monotonic system.

In Section 4, we explained why removing the ambiguity between Shift and Right-Arcs altogether was an inferior strategy. Failing to discriminate between arcs reachable by monotonic and non-monotonic paths does just that, so this oracle did not perform well in preliminary experiments on the development data.

Instead, we want to learn a model that will offer its best prediction of Shift vs. Right-Arc, which we expect to usually be correct. However, in those cases where the model does make the wrong decision, it should have the ability to later over-turn that decision, by having an unconstrained choice of Reduce vs. Left-Arc.

In order to correct for that, we don't use the non-monotonic oracles directly when training the parser, but instead train the parser using both the monotonic and non-monotonic oracles simultaneously by combining their judgements: while we always prefer zero-cost non-monotonic actions to monotonic-actions with non-zero cost, if the non-monotonic oracle assigns several actions a zero-cost, we prefer to follow those actions that are also assigned a zero-cost by the monotonic oracle, as these actions lead to the best outcome without relying on a non-monotonic (repair) operation down the road.

## 7 Experiments

We base our experiments on the parser described by Goldberg and Nivre (2012). We began by implementing their baseline system, a standard Arc-Eager parser using an averaged Perceptron learner and the extended feature set described by Zhang and Nivre (2011). We follow Goldberg and Nivre (2012) in training all models for 15 iterations, and shuffling the sentences before each iteration.

Because the sentence ordering affects the model's accuracy, all results are averaged from scores produced using 20 different random seeds. The seed determines how the sentences are shuffled before each iteration, as well as when to follow an optimal action and when to follow a non-optimal action during training. The Wilcoxon signed-rank test was used for significance testing.

A train/dev/test split of 02-21/22/23 of the Penn Treebank WSJ (Marcus et al., 1993) was used for all models. The data was converted into Stanford dependencies (de Marneffe et al., 2006) with copula-as-head and the original PTB noun-phrase bracketing. We also evaluate our models on dependencies created by the PENN2MALT tool, to assist comparison with previous results. Automatically assigned POS tags were used during training, to match the test data more closely. [6] We also evaluate the non-monotonic transitions on the CoNLL 2007 multi-lingual data.

| | Stanford | | MALT | |
|---|---|---|---|---|
| | W | S | W | S |
| | Unlabelled Attachment | | | |
| Baseline (G&N-12) | 91.2 | 42.0 | 90.9 | 39.7 |
| NM L | 91.4 | 43.1 | 91.0 | 40.1 |
| NM D | 91.4 | 42.8 | 91.1 | 41.2 |
| NM L+D | 91.6 | 43.3 | 91.3 | 41.5 |
| | Labelled Attachment | | | |
| Baseline (G&N-12) | 88.7 | 31.8 | 89.7 | 36.6 |
| NM L | 89.0 | 32.5 | 89.8 | 36.9 |
| NM D | 88.9 | 32.3 | 89.9 | 37.7 |
| NM L+D | 89.1 | 32.7 | 90.0 | 37.9 |

Table 1: Development results on WSJ 22. Both non-monotonic transitions bring small improvements in per-token (W) and whole sentence (S) accuracy, and the improvements are additive.

## 8 Results and analysis

Table 1 shows the effect of the non-monotonic transitions on labelled and unlabelled attachment score on the development data. All results are averages from 20 models trained with different random seeds, as the ordering of the sentences at each iteration of the Perceptron algorithm has an effect on the system's accuracy. The two non-monotonic transitions each bring small but statistically significant improvements that are additive when combined in the NM L+D system. The result is stable

[5] If using a labeled reduce transition, the label assignment costs should be handled here.

[6] We thank Yue Zhang for supplying the POS-tagged files used in the Zhang and Nivre (2011) experiments.

across both dependency encoding schemes.

**Frequency analysis.** Recall that there are two pop moves available: Left-Arc and Reduce. The Left-Arc is considered non-monotonic if the top of the stack has a head specified, and the Reduce move is considered non-monotonic if it does not. How often does the parser select monotonic and non-monotonic pop moves, and how often is its decision correct?

In Table 2, the True Positive column shows how often non-monotonic transitions were used to add gold standard dependencies. The False Positive column shows how often they were used incorrectly. The False Negative column shows how often the parser missed a correct non-monotonic transition, and the True Negative column shows how often the monotonic alternative was correctly preferred (e.g. the parser correctly chose monotonic Reduce in place of non-monotonic Left-Arc). Punctuation dependencies were excluded.

The current system has high precision but low recall for repair operations, as they are relatively rare in the gold-standard. While we already see improvements in accuracy, the upper bound achievable by the non-monotonic operations is higher, and we hope to approach it in the future using improved learning techniques.

**Linguistic analysis.** To examine what constructions were being corrected, we looked at the frequencies of the labels being introduced by the non-monotonic moves. We found that there were two constructions being commonly repaired, and a long-tail of miscellaneous cases.

The most frequent repair involved the *mark* label. This is assigned to conjunctions introducing subordinate clauses. For instance, in the sentence *Results were released after the market closed*, the Stanford scheme attaches *after* to *closed*. The parser is misled into greedily attaching *after* to *released* here, as that would be correct if *after* were a preposition, as in *Results were released after midnight*. This construction was repaired 33 times, 13 where the initial decision was *mark*, and 21 times the other way around. The other commonly repaired construction involved greedily attaching an object that was actually the subject of a complement clause, e.g. *NCNB corp. reported net income doubled*. These were repaired 19 times.

**WSJ evaluation.** Table 3 shows the final test results. While still lagging behind search based parsers, we push the boundaries of what can be achieved with a purely greedy system, with a statistically significant improvement over G&N 12.

| | TP | FP | TN | FN |
|---|---|---|---|---|
| Left-Arc | 60 | 14 | 18,466 | 285 |
| Reduce | 52 | 26 | 14,066 | 250 |
| Total | 112 | 40 | 32,532 | 535 |

Table 2: True/False positive/negative rates for the prediction of the non-monotonic transitions. The non-monotonic transitions add correct dependencies 112 times, and produce worse parses 40 times. 535 opportunities for non-monotonic transitions were missed.

| System | $O$ | Stanford | | Penn2Malt | |
|---|---|---|---|---|---|
| | | LAS | UAS | LAS | UAS |
| K&C 10 | $n^3$ | — | — | — | 93.00 |
| Z&N 11 | $nk$ | 91.9 | 93.5 | 91.8 | 92.9 |
| G&N 12 | $n$ | 88.72 | 90.96 | — | — |
| Baseline(G&N-12) | $n$ | 88.7 | 90.9 | 88.7 | 90.6 |
| NM L+D | $n$ | 88.9 | 91.1 | 88.9 | 91.0 |

Table 3: WSJ 23 test results, with comparison against the state-of-the-art systems from the literature of different runtimes. **K&C 10**=Koo and Collins (2010); **Z&N 11**=Zhang and Nivre (2011); **G&N 12**=Goldberg and Nivre (2012).

**CoNLL 2007 evaluation.** Table 4 shows the effect of the non-monotonic transitions across the ten languages in the CoNLL 2007 data sets. Statistically significant improvements in accuracy were observed for five of the ten languages. The accuracy improvement on Hungarian and Arabic did not meet our significance threshold. The non-monotonic transitions did not decrease accuracy significantly on any of the languages.

## 9 Related Work

One can view our non-monotonic parsing system as adding "repair" operations to a greedy, deterministic parser, allowing it to undo previous decisions and thus mitigating the effect of incorrect parsing decisions due to uncertain future, which is inherent in greedy left-to-right transition-based parsers. Several approaches have been taken to address this problem, including:

**Post-processing Repairs** (Attardi and Ciaramita, 2007; Hall and Novák, 2005; Inokuchi and Yamaoka, 2012) Closely related to stacking, this line of work attempts to train classifiers to repair attachment mistakes after a parse is proposed by a parser by changing head attachment decisions. The present work differs from these by incorporating the repair process into the transition system.

**Stacking** (Nivre and McDonald, 2008; Martins et al., 2008), in which a second-stage parser runs over the sentence using the predictions of the first parser as features. In contrast our parser works in

| System | AR | BASQ | CAT | CHI | CZ | ENG | GR | HUN | ITA | TUR |
|---|---|---|---|---|---|---|---|---|---|---|
| Baseline | 83.4 | 76.2 | 91.5 | 82.3 | 78.8 | 87.9 | 81.2 | 77.6 | 83.8 | 78.0 |
| NM L+D | *83.6* | 76.1 | 91.5 | **82.7** | **80.1** | **88.4** | **81.8** | *77.9* | **84.1** | 78.0 |

Table 4: Multi-lingual evaluation. Accuracy improved on Chinese, Czech, English, Greek and Italian ($p < 0.001$), trended upward on Arabic and Hungarian ($p < 0.005$), and was unchanged on Basque, Catalan and Turkish ($p > 0.4$).

a single, left-to-right pass over the sentence.

**Non-directional Parsing** The EasyFirst parser of Goldberg and Elhadad (2010) tackles similar forms of ambiguities by dropping the Shift action altogether, and processing the sentence in an easy-to-hard bottom-up order instead of left-to-right, resulting in a greedy but non-directional parser. The indeterminate processing order increases the parser's runtime from $O(n)$ to $O(n \log n)$. In contrast, our parser processes the sentence incrementally, and runs in a linear time.

**Beam Search** An obvious approach to tackling ambiguities is to forgo the greedy nature of the parser and instead to adopt a beam search (Zhang and Clark, 2008; Zhang and Nivre, 2011) or a dynamic programming (Huang and Sagae, 2010; Kuhlmann et al., 2011) approach. While these approaches are very successful in producing high-accuracy parsers, we here explore what can be achieved in a strictly deterministic system, which results in much faster and incremental parsing algorithms. The use of non-monotonic transitions in beam-search parser is an interesting topic for future work.

## 10 Conclusion and future work

We began this paper with the observation that because the Arc-Eager transition system (Nivre et al., 2004) attaches a word to its governor either when the word is pushed onto the stack or when it is popped off the stack, monotonicity (plus the "tree constraint" that a word has exactly one governor) implies that a word's push-move determines its associated pop-move. In this paper we suggest relaxing the monotonicity constraint to permit the pop-move to alter existing attachments if appropriate, thus breaking the 1-to-1 correspondence between push-moves and pop-moves. This permits the parser to correct some early incorrect attachment decisions later in the parsing process. Adding additional transitions means that in general there are multiple transition sequences that generate any given syntactic analysis, i.e., our non-monotonic transition system generates spurious ambiguities (note that the Arc-Eager transition system on its own generates spurious ambiguities). As we explained in the paper, with the greedy decoding used here additional spurious ambiguity is not necessarily a draw-back.

The conventional training procedure for transition-based parsers uses a "static" oracle based on "gold" parses that never predicts a non-monotonic transition, so it is clearly not appropriate here. Instead, we use the incremental error-based training procedure involving a "dynamic" oracle proposed by Goldberg and Nivre (2012), where the parser is trained to predict the transition that will produce the best-possible analysis from its current configuration. We explained how to modify the Goldberg and Nivre oracle so it predicts the optimal moves, either monotonic or non-monotonic, from any configuration, and use this to train an averaged perceptron model.

When evaluated on the standard WSJ training and test sets we obtained a UAS of 91.1%, which is a 0.2% improvement over the already state-of-the-art baseline of 90.9% that is obtained with the error-based training procedure of Goldberg and Nivre (2012). On the CoNLL 2007 datasets, accuracy improved significantly on 5/10 languages, and did not decline significantly on any of them.

Looking to the future, we believe that it would be interesting to investigate whether adding non-monotonic transitions is beneficial in other parsing systems as well, including systems that target formalisms other than dependency grammars. As we observed in the paper, the spurious ambiguity that non-monotonic moves introduce may well be an advantage in a statistical parser with an enormous state-space because it provides multiple pathways to the correct analysis (of which we hope at least one is navigable).

We investigated a very simple kind of non-monotonic transition here, but of course it's possible to design transition systems with many more transitions, including transitions that are explicitly designed to "repair" characteristic parser errors. It might even be possible to automatically identify the most useful repair transitions and incorporate them into the parser.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This research was supported under the Australian Research Council's Discovery Projects funding scheme (project numbers DP110102506 and DP110102593).

## References

Stephen Abney and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20(3):233–250.

Giuseppe Attardi and Massimiliano Ciaramita. 2007. Tree revision learning for dependency parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 388–395. Association for Computational Linguistics, Rochester, New York.

Miguel Ballesteros and Joakim Nivre. 2013. Going to the roots of dependency parsing. *Computational Linguistics. 39:1.*

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC).*

Lyn Frazier and Keith Rayner. 1982. Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology*, 14(2):178–210.

Yoav Goldberg and Michael Elhadad. 2010. 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 (NAACL HLT)*, pages 742–750.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (Coling 2012)*. Association for Computational Linguistics, Mumbai, India.

Keith Hall and Vaclav Novák. 2005. Corrective modeling for non-projective dependency parsing. In *Proceedings of the 9th International Workshop on Parsing Technologies (IWPT)*, pages 42–52.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1077–1086.

Akihiro Inokuchi and Ayumu Yamaoka. 2012. Mining rules for rewriting states in a transition-based dependency parser for English. In *Proceedings of COLING 2012*, pages 1275–1290. The COLING 2012 Organizing Committee, Mumbai, India.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–11.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 673–682. Association for Computational Linguistics, Stroudsburg, PA, USA.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330.

André Filipe Martins, Dipanjan Das, Noah A. Smith, and Eric P. Xing. 2008. Stacking dependency parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 157–166.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160.

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

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In

Hwee Tou Ng and Ellen Riloff, editors, *HLT-NAACL 2004 Workshop: Eighth Conference on Computational Natural Language Learning (CoNLL-2004)*, pages 49–56. Association for Computational Linguistics, Boston, Massachusetts, USA.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 950–958.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press, Cambridge, MA.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571. Association for Computational Linguistics, Honolulu, Hawaii.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193. Association for Computational Linguistics, Portland, Oregon, USA.