Exploring beam-based shift-reduce dependency parsing with DyALog: Results from the SPMRL 2013 shared task

Éric Villemonte de La Clergerie

Exploring beam-based shift-reduce dependency parsing with DyALog: Results from the SPMRL 2013 shared task

Éric Villemonte de la Clergerie
INRIA - Rocquencourt - B.P. 105
78153 Le Chesnay Cedex, FRANCE
Eric.De_La_Clergerie@inria.fr

Abstract

The SPMRL 2013 shared task was the opportunity to develop and test, with promising results, a simple beam-based shift-reduce dependency parser on top of the tabular logic programming system DYALOG. The parser was also extended to handle ambiguous word lattices, with almost no loss w.r.t. disambiguated input, thanks to specific training, use of oracle segmentation, and large beams. We believe that this result is an interesting new one for shift-reduce parsing.

1 Introduction

DYALOG is a tabular-based logic programming environment, including a language (variant of Prolog), a bootstrapped compiler, and C-based abstract machine. It is mostly used for chart-like parsing (de La Clergerie, 2005b), in particular for a wide coverage French Tree Adjoining Grammar (de La Clergerie, 2005a). However, DYALOG offers all the power of a programming language a la Prolog, with some specific advantages, and it was tempting to try it on statistical parsing paradigms. The SPMRL 2013 shared task (Seddah et al., 2013) was an interesting opportunity to develop a simple (non-deterministic) beam-based shift-reduce dependency parser, called DYALOG-SR, inspired by (Huang and Sagae, 2010).

The main advantage of logic programming is the (almost) transparent handling of non-determinism, useful for instance to handle ambiguous word lattices. DYALOG allows an easy tabulation of items, and their fast retrieval (thanks to full term indexing), needed for the dynamic programming part of the algorithm. Thanks to structure sharing and term hashing, it also reduces the costs related to the tabulation of multiple items (sharing subparts) and to term unification. Logic programs tend to be very concise, with, in our case, around 1500 lines of DYALOG code. However, one of the disadvantages of (pure) logic programming, and of DYALOG in particular, is the handling of mutable structures, which motivated the development of a companion C module (around 850 lines) to handle statistical models (loading, querying, updating, and saving).

We briefly present the implemented algorithm (Section 2) and list the preliminary adaptations done for the 9 languages of the shared task (Section 3). We analyze in Section 4 the official results for DYALOG-SR. Recent developments corrected some weaknesses of DYALOG-SR. In particular, we explain in Section 5 how we seriously improved the parsing of ambiguous lattices, an important new result for shift-reduce parsing. Finally, Section 6 provides some empirical data about the efficiency and complexity of the algorithm.

2 A Dynamic Programming Shift-Reduce parser

We used (Huang and Sagae, 2010) as the starting point for this work, in particular using the same simple *arc-standard* strategy for building projective dependency trees, defined by the deductive system of Figure 1. In a configuration $m{:}\langle j, S\rangle{:}c$, m denotes the number of transitions applied since the axiom configuration, j the current position in the input string, S the stack of partial dependency trees built so far, and c the cost. A *shift* transition pushes the next input symbol on top of the stack while the two *reduce* transitions combine the 2 topmost stack trees, add a new (labeled) leftmost or rightmost de-

pendency edge between their roots, and remove the newly governed subtree from the stack. The delta cost ξ, λ, and ρ denote the cost of each operation w.r.t. the input configuration.

$$\begin{array}{ll} \text{input:} & w_0 \dots w_{n-1} \\ \text{axiom} & 0{:}\langle 0, \epsilon\rangle{:}0 \\ & \text{shift}\dfrac{m{:}\langle j, S\rangle{:}c}{m+1{:}\langle j+1, S|w_j\rangle{:}c+\xi} \\ & re_{_l\curvearrowleft}\dfrac{m{:}\langle j, S|s_1|s_0\rangle{:}c}{m+1{:}\langle j, S|s_1\ {}_l\!\curvearrowleft\! s_0\rangle{:}c+\lambda} \\ & re_{\curvearrowright_l}\dfrac{m{:}\langle j, S|s_1|s_0\rangle{:}c}{m+1{:}\langle j, S|s_1\ \curvearrowright_l\ s_0\rangle{:}c+\rho} \\ \text{goal} & 2n-1{:}\langle n, s_0\rangle{:}c \end{array}$$

Figure 1: Arc-standard deductive system

From the configurations, the deductive system, and the configuration elements used to determine the transition costs, it is relatively straightforward to design *items* denoting partial configurations standing for equivalence classes of configurations and allowing computation sharing, following the principle of *Dynamic Programming*. The deduction rules are adapted to work on items and beam search (with size b) is then achieved by keeping only the b best items for each step m[1]. By following *backpointers* from items to parents, it is possible to retrieve the best transition sequence and the best dependency tree.

```
item{ step => M,
      right => J,
      stack => S0,  % topmost trees
      stack1 => S1, %
      prefix => Cost,  % max cost
      inside => ICost  % inside cost
    }.
back(Item,Action,Parent1,Parent2,C).
tail(Item,Ancestor).
```

Listing 1: Item structure

Instead of the items proposed in (Huang and Sagae, 2010), we switched to items closer to those proposed in (Goldberg et al., 2013), corresponding to *Tree Structured Stacks* (TSS), where stack tails are shared among items, as defined by Listing 1. The prefix cost corresponds to the maximal cost attached to the item, starting from the initial item. The inside cost is the maximal cost for a derivation from some ancestor item where s_0 was shifted on the stack, and is used to adjust the total cost for different ancestor items. The items are completed by backpointers (using asserted facts `back/5`) and links to the potential stack tails (using asserted facts `tail/2`) needed to retrieve the lower part of a stack when applying a reduce action. Figure 2 shows the adaptation for items of some of the deductive rules.

$$\text{shift}\frac{\begin{array}{c} I = m{:}\langle j, s_0, s_1\rangle{:}(c, \iota) \end{array}}{\begin{array}{c} J = m+1{:}\langle j+1, w_j, s_0\rangle{:}(c+\xi, \xi) \\ \text{tail}(J) \mathrel{+}= I \\ \text{back}(J) \mathrel{+}= (\text{shift}, I, \text{nil}, c+\xi) \end{array}}$$

$$re_{_\iota\curvearrowleft}\frac{\begin{array}{c} I = m{:}\langle j, s_0, s_1\rangle{:}(c, \iota) \\ J = \_{:}\langle \_, s_1, s_2\rangle{:}(c', \iota') \in \text{tail}(I) \end{array}}{\begin{array}{c} K = m+1{:}\langle j, s_1\ {}_l\!\curvearrowleft\! s_0, s_2\rangle{:}(c'+\delta, \iota'+\delta) \\ \delta = \iota + \lambda \\ \text{tail}(K) \cup\!= \text{tail}(J) \\ \text{back}(K) \mathrel{+}= ({}_l\!\curvearrowleft, I, J, c'+\delta) \end{array}}$$

Figure 2: Deductive system on items (fragment)

The stack elements for configuration are dependency trees, but approximations can be used for the item fields `stack` and `stack1`, under the condition that sufficient information remains to apply the transitions and to compute the costs. In practice, we keep information about the root node, and, when present, the leftmost and rightmost dependency edges, the numbers of left and right dependencies (*valency*), and the label sets (*domain*) for the left and right dependencies.

The training phase relies on sequences of actions provided by an oracle and uses a simple *averaged structured perceptron* algorithm (Daume, 2006). The underlying statistical model is updated positively for the actions of the oracle and negatively for the actions of the parser, whenever a point of divergence is found. Several updating strategies may be considered (Huang et al., 2012), and, in our case, we update as early (*early update*) and as often as possible: after completion of Step $m+1$, we update the model locally (i.e. for the last action) whenever

[1] Because we use Dynamic Programming techniques, keeping the b-best items at step m actually corresponds to keep more than the b-best configurations at step m.

- the best item B^O_{m+1} derived from the oracle item O_m at Step m differs from the expected oracle item O_{m+1};
- the oracle item O_{m+1} is not in the beam, for intermediary steps $m < 2n - 2$;
- the oracle item O_{m+1} is not the best item, for the last step $m = 2n - 2$.

We use a relatively standard set of *word features* related to the CONLL fields such as `lex` (FORM), `lemma`, `cat` (CPOSTAG), `fullcat` (POSTAG), `mstag` (morphosyntactic features FEATS). They apply to the next unread word (`*I`, say `lemmaI`), the two next lookahead words (`*I2` and `*I3`), and (when present) to the 2 stack root nodes (`*0` and `*1`), their leftmost and rightmost child (*before* `b*[01]` and *after* `a*[01]`). We have *dependency features* such as the labels of the leftmost and rightmost edges (`[ab]label[01]`), the left and right valency and domains (`[ab][vd][01]`). Finally, we have 3 (discretized) *distance features* between the next word and the stack roots (`delta[01]`) and between the two stack roots (`delta01`). Most feature values are atomic (either numerical or symbolic), but they can also be (recursively) a list of values, for instance for the `mstag` and *domain* features.

A tagset (for a given language and/or treebank) contains a set of *feature templates*, each template being a sequence of features (for instance `fullcat0:fullcat1:blabel0`).

Model management is a key factor for the efficiency of the algorithm, both for querying or updating the costs attached to a configuration. Therefore, we developed a specialized C companion module. A model is represented by a hash trie to factor the prefixes of the templates. Costs are stored in the leaves (for selecting the labels) and their immediate parent (for selecting between the `shift` and `reduce` base actions), ensuring join learning with smoothing of an action and a label. Querying is done by providing a tree-structured argument representing the feature values for all templates[2], with the possibility to leave underspecified the action and the label. By traversing in a synchronous way the model trie and the argument tree, and accumulating costs for all possible actions and labels, a single query returns in order the cost for the b best actions. Furthermore, when a feature value is a list, the traversal is run for all its components (with summation of all found costs).

3 Preparing the shared task

We trained the parser on the training and dev dependency treebanks kindly provided by the organizers for the 9 languages of the task, namely Arabic[3], Basque (Aduriz et al., 2003), French (Abeillé et al., 2003), German (Brants et al., 2002; Seeker and Kuhn, 2012), Hebrew (Sima'an et al., 2001; Tsarfaty, 2013; Goldberg, 2011), Hungarian (Vincze et al., 2010; Csendes et al., 2005), Korean (Choi et al., 1994; Choi, 2013) , Polish (Świdziński and Woliński, 2010), Swedish (Nivre et al., 2006).

Being very short in time, we essentially used the same set of around 110 templates for all languages. Nevertheless, minimal tuning was performed for some languages and for the pred data mode (when using predicted data), as summarized below.

For French, the main problem was to retrieve MWEs (*Multi Word Expression*) in `pred` data mode. Predicted features `mwehead` and `pred` were added, thanks to a list of MWEs collected in the gold treebank and in the French lexicon LEFFF (Sagot et al., 2006). We also added the predicted feature `is_number` to help detecting numerical MWEs such as *120 000*, and also a `is_capitalized` feature. For all data modes, we added a subcategorization feature for verbs (with a list value), again extracted from LEFFF.

For Arabic, Hebrew, and Swedish, the `lemma` feature is removed because of the absence of lemma in the treebanks. Similarly, for Polish and German, with identical CPOS and POS tagsets, we remove the `cat` feature.

For Hungarian, the `SubPOS` morphosyntactic feature is appended to the fullcat feature, to get a

[2]The tree structure of the argument mirrors the tree structure of the templates and getting the argument tree for a configuration is actually a fast and very low memory operation, thanks to unification and structure sharing.

[3]We used the shared task Arabic data set, originally provided by the LDC (Maamouri et al., 2004), specifically its SPMRL 2013 dependency instance, derived from the Columbia Catib Treebank (Habash and Roth, 2009; Habash et al., 2009)

richer set of POS. The set of dependency labels being large (450 labels), we split the labels into lists of more elementary ones for the `label` features.

Similarly, the Korean POS tags are also split into lists, because of their large number (2743 tags) and of their compound structure.

For French, Hebrew, and Korean, in order to compensate initially large differences in performance between the gold and pred modes, we added, for the pred mode, `dict` features filled by predicted information about the possible tags for a given form, thanks to the `dict` lexicons provided by the IMS_SZEGED team.

Finally, we discovered very late that the dependency trees were not necessarily projective for a few languages. A last-second solution was to use the MALT projectivization / deprojectivization wrappers (Nivre and Nilsson, 2005) to be able to train on projectivized versions of the treebanks for German, Hungarian, and Swedish, while returning non projective trees.

4 First results

Under the team label `ALPAGE-DYALOG`, we have returned parsed data for the 9 languages of the shared task, for the **full** and **5k** training size modes, and for the **gold** and **pred** data modes. For each configuration, we provided 3 runs, for beam sizes 8, 6, and 4. The results are synthesized in Tables 2, with LAS[4] on the `test` and `dev` files, contrasted with the LAS for the best system, the baseline, and the mean LAS of all systems. The tables show that DYALOG-SR cannot compete with the best system (like most other participants !), but performs reasonably well w.r.t. the baseline and the mean LAS of the participants, at least in the **gold/full** case.

The system is proportionally less accurate on smaller training treebanks (**5k** case), lacking good smoothing mechanisms to deal with data sparseness. The **pred** case is also more difficult, possibly again because of data sparseness (less reliable information not compensated by bigger treebanks) but also because we exploited no extra information for some languages (such as Basque or Swedish).

The big drop for German in `pred/5k` case comes from the fact we were unable to deprojectivize the parsed test file with Malt[5] and returned data built using an old model not relying on Malt proj/deproj wrappers.

For Hungarian, a possible reason is the high level of multiple roots in sentences, not compatible with our initial assumption of a single root per sentence. New experiments, after modifying slightly the algorithm to accept multiple roots[6], confirm this hypothesis for Hungarian, and for other languages with multiple roots, as shown in Table 1.

language	#roots/sent	single	multiple
Hungarian	2.00	79.22	82.90
Arabic	1.21	87.17	87.71
Basque	1.21	81.09	82.28
German	1.09	90.95	91.29

Table 1: Taking into account multiple roots (on gold/full)

Finally, the Korean case, where we are below the baseline, remains to be explained. For the **pred** case, it could come from the use of the **KAIST** tagset instead of the alternative **Seijong** tagset. For the **gold** case, the results for all participants are actually relatively close.

5 Handling ambiguous lattices

One of the important and innovative sub-tasks of the SPMRL campaign was to parse ambiguous lattices using statistical methods. A *word lattice* is just a Directed Acyclic Graph (DAG) whose edges are decorated by words with their features and whose nodes denote positions in the sentence, as represented in Figure 3 for an Hebrew sentence. A valid analysis for a sentence should follow a path in the DAG from its root node at position 0 till its final node at position n. Each edge may be associated with an unique identifier to be able to refer it.

Lattice parsing is rather standard in chart-parsing[7] and since the beginning, thanks to DYALOG's support, DYALOG-SR was designed to parse ambiguous word lattices as input, but originally using

[4]Labeled Attachment Score, with punctuation being taking into account.

[5]because of non-termination on at least one sentence.

[6]Essentially, the initial configuration becomes $0{:}\langle 0, \mathbf{0}\rangle{:}0$ and the final one $2n{:}\langle n, \mathbf{0} \curvearrowright \star\rangle{:}c$ using $\mathbf{0}$ as a virtual root node.

[7]being formalized as computing the intersection of a grammar with a regular language.

language	DYALOG-SR test	dev	b	other systems best	baseline	mean
Arabic	85.87	86.99	4	89.83	82.28	**86.11**
Basque	**80.39**	81.09	6	86.68	69.19	79.58
French	**87.69**	87.94	8	90.29	79.86	85.99
German	**88.25**	90.89	6	91.83	79.98	86.80
Hebrew	**80.70**	81.31	8	83.87	76.61	80.13
Hungarian	79.60	79.09	4	88.06	72.34	**81.36**
Korean	88.23	89.24	6	89.59	**88.43**	88.91
Polish	**86.00**	86.94	8	89.58	77.70	83.79
Swedish	**79.80**	75.94	6	83.97	75.73	79.21

(a) gold/full

language	DYALOG-SR test	dev	b	other systems best	baseline	mean
Arabic	83.25	84.24	8	87.35	80.36	**83.79**
Basque	**79.11**	79.03	8	85.69	67.13	78.33
French	**85.66**	0.00	8	88.73	78.16	84.49
German	**83.88**	87.21	6	87.70	76.64	83.06
Hebrew	**80.70**	81.31	8	83.87	76.61	80.13
Hungarian	78.42	79.09	4	87.21	71.27	**80.42**
Korean	81.91	84.50	6	83.74	**81.93**	82.74
Polish	**85.67**	0.00	8	89.16	76.64	83.13
Swedish	**79.80**	0.00	6	83.97	75.73	79.21

(b) gold/5k

language	DYALOG-SR test	dev	b	other systems best	baseline	mean
Arabic	81.20	82.18	8	86.21	80.36	**82.57**
Basque	77.55	78.47	4	85.14	70.11	**79.13**
French	**82.06**	82.88	8	85.86	77.98	81.03
German	**84.80**	88.38	8	89.65	77.81	84.33
Hebrew	**73.63**	74.74	6	80.89	69.97	73.30
Hungarian	75.58	75.74	6	86.13	70.15	**79.23**
Korean	81.02	82.45	6	86.62	**82.06**	83.09
Polish	**82.56**	83.87	8	87.07	75.63	81.40
Swedish	77.54	73.37	8	82.13	73.21	**77.65**

(c) pred/full

language	DYALOG-SR test	dev	b	other systems best	baseline	mean
Arabic	78.65	79.25	8	83.66	78.48	**80.19**
Basque	76.06	76.11	6	83.84	68.12	**77.76**
French	**80.11**	0.00	4	83.60	76.54	79.31
German	73.07	84.69	8	85.08	**74.81**	79.34
Hebrew	**73.63**	74.74	6	80.89	69.97	73.30
Hungarian	74.48	75.55	6	85.24	69.08	**78.31**
Korean	73.79	76.66	6	80.80	**74.87**	76.34
Polish	**82.04**	0.00	8	86.69	75.29	80.96
Swedish	77.54	72.44	8	82.13	73.21	**77.65**

(d) pred/5k

Table 2: Official results

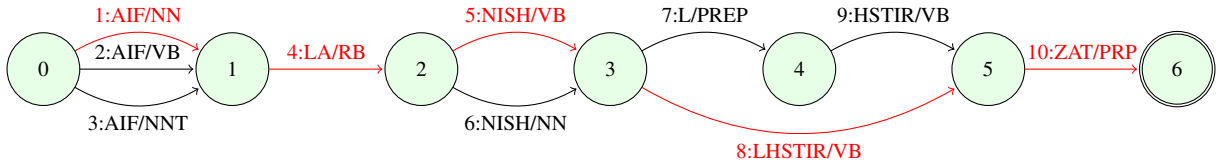

Figure 3: An ambiguous Hebrew word lattice (with gold segmentation path *AIF LA NISH LHSTIR ZAT*)

models trained on standard CONLL non ambiguous sentences. However, the initial experiments with Hebrew lattices (Table 3, using TED metric) have shown an important drop of 11 points between non ambiguous lattices (similar to standard CONLL files) and ambiguous ones.

	Hebrew disamb	Hebrew nodisamb	Arabic disamb
no training	87.34	76.35	87.32
spec. training		86.75	

Table 3: Results on dev lattices (TED accuracy $*$ 100)

The main reason for that situation is that multiple paths of various lengths are now possible when traversing a lattice. Final items are no longer associated with the same number of steps ($2n-1$) and final items with a large number of steps (corresponding to longest paths in the lattice) tend to be favored over those with a small number of steps (corresponding to shortest paths), because the transition costs tend to be positive in our models.

A first attempt to compensate this bias was to "*normalize*" path lengths by adding (incrementally) some extra cost to the shortest paths, proportional to the number of missing steps. Again using models trained on non-ambiguous segmentations, we gained around 3 points (TED accuracy around 79) using this approach, still largely below the non-ambiguous case.

Finally, we opted for specific training on lattice, with the idea of introducing the new `length` word feature, whose value is defined, for a word, as the difference between its right and left position in the lattice. To exploit this feature, we added the following 9 templates: `length[I,I2,0]`,

`fullcat[I,I2,0]:length[I,I2,0]`, `lengthI:lengthI2`, `length0:lengthI`, and `length0:lengthI:lengthI2`.

Then, to ensure that we follow valid lattice paths, the configurations and items were completed with three extra *lookahead* fields `la[123]` to remember the edge identifiers of the lookahead words that were consulted. Obviously, adding this extra information increases the number of items, only differing on their lookahead sequences, but it is an important element for the coherence of the algorithm.

The reduce actions are kept unchanged, modulo the propagation without change of the lookahead identifiers, as shown below:

$$re_{l\curvearrowleft}\frac{m :< j, S|s_1|s_0, \mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3 >: c}{m+1 :< j, S|s_1\ {}_l\!\curvearrowleft s_0, \mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3 >: c+\lambda}$$

$$re_{\curvearrowright_l}\frac{m :< j, S|s_1|s_0, \mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3 >: c}{m+1 :< j, S|s_1 \curvearrowright_l s_0, \mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3 >: c+\rho}$$

On the other hand, the shift action consumes its first lookahead identifier la_1 (for a word between position j and k) and selects a new lookahead identifier la_4 (which must be a valid choice for continuing the path $\mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3$):

$$\mathrm{shift}\frac{m :< j, S, \mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3 >: c}{m+1 :< k, S|\mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3, \mathrm{la}_4 >: c+\xi}$$

It should be noted that for a given position j in the lattice, we may have several items only differing by their lookahead sequences $\mathrm{la}_1, \mathrm{la}_2, \mathrm{la}_3$, and each of them will produce at least one new item by shifting la_1, and possibly more than one because of multiple la_4. However, several of these new shifted items are discarded because of the beam. Learning good estimations for the shift actions becomes a key point, more important than for usual shift-reduce algorithms.

In order to do that, we modified the oracle to provide information about the *oracle segmentation path* in the lattice, essentially by mentioning which edge identifier should be used for each oracle shift action. It should be noted that this information is also sufficient to determine the lookahead sequence for each oracle item, and in particular, the new edge identifier la_4 to be retrieved for the shift actions.

An issue was however to align the predicted lattices with the gold sentences (implementing a standard dynamic programming algorithm) in order to find the oracle segmentation paths. Unfortunately, we found that the segmentation path was missing for 1,055 sentences in the provided Hebrew lattices (around 20% of all sentences). Rather than discarding these sentences from an already small training set, we decided to keep them with incomplete prefix segmentation paths and oracles.

Figure 4 shows the strong impact of a specific training and of using large beams, with a TED accuracy climbing up to 86.75 (for beam size 16), close to the 87.34 reached on non-ambiguous lattices (for beam 6). Increasing beam size (around 3 times) seems necessary, probably for compensating the lattice ambiguities (2.76 transitions per token on average). However, even at beam=6, we get much better results (TED=83.47) than without specific training for the same beam size (TED=76.35).

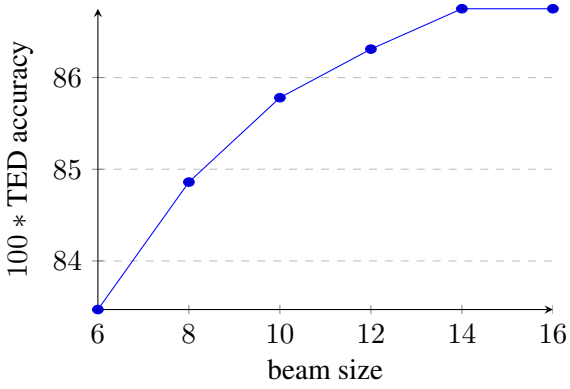

Figure 4: Score on Hebrew lattices w.r.t. beam size

To test the pertinence of the `length` features, we did some training experiments without these features. Against our expectations, we observed only a very low drop in performance (TED 86.50, loss $=$ 0.25). It is possible that the `lex` features are sufficient, because only a relatively restricted set of (frequent) words have segmentations with length > 1. In practice, for the Hebrew 5k training lattices, we have 4,141 words with length > 1 for 44,722 occurrences (22.21% of all forms, and 12.65% of all occurrences), with around 80% of these occurrences covered by only 1,000 words. It is also possible that we under-employ the `length` features in too few templates, and that larger gains could be obtained.

6 Empirical analysis

The diversity and amount of data provided for the shared task was the opportunity to investigate more closely the properties of DYALOG-SR, to identify its weaknesses, and to try to improve it.

The usefulness of beams has been already proved in the case of Hebrew ambiguous lattices, and Figure 5 confirms that, in general, we get serious improvements using a beam, but in practice, beam sizes above 8 are not worth it. However, we observe almost no gain for Korean, a situation to be investigated.

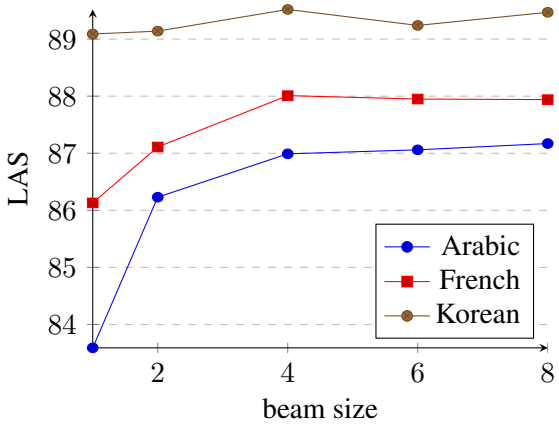

Figure 5: Accuracy evolution w.r.t. beam size

Efficiency was not the main motivation for this work and for the shared task. However, it is worthwhile to examine the empirical complexity of the algorithm w.r.t. beam size and w.r.t. sentence length. As shown in Figure 6, the average speed at beam=1 is around 740 tokens by second. At best, we expect a linear decreasing of the speed w.r.t. to beam size, motivating the use of a normalized speed by multiplying by the size. Surprisingly, we observe a faster normalized speed than expected for small beam sizes, maybe arising from computation sharing. However, for larger beam sizes, we observe a strong decrease, maybe related to beam management through (longer) sorted DYALOG lists, but also to some limits of term indexing[8]. The same experience carried for large beam sizes on the Hebrew lattices does not exhibit the same degradation, a point to be investigated but which suggests some kind of equivalence between beam=4 on non ambiguous input string and beam=12 on ambiguous lattices (also reflected in accuracy evolution).

[8]Even with efficient term indexing, checking the presence of an item in DYALOG table is not a constant time operation.

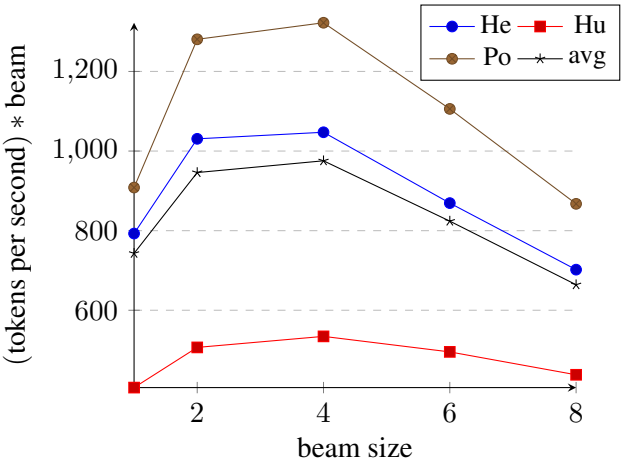

Figure 6: Normalized speed w.r.t. beam size (dev)

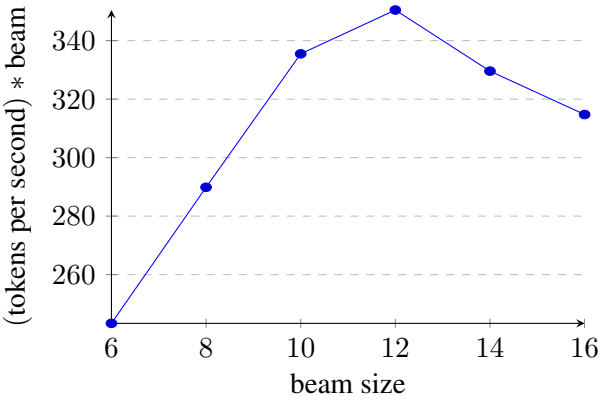

Figure 7: Normalized speed w.r.t. beam size (lattices)

Collecting parsing times for the sentences under length 80 from all training files and for all training iterations, Figure 8 confirms that parsing time (divided by beam size) is linear w.r.t. sentence length both for beam=1 and beam=8. On the other hand, we observe, Figure 9, that the number of updates increases with beam size (confirming that larger beams offer more possibilities of updates), but also non linearly with sentence length.

7 Conclusion

We have presented DYALOG-SR, a new implementation on top of DYALOG system of a beam-based

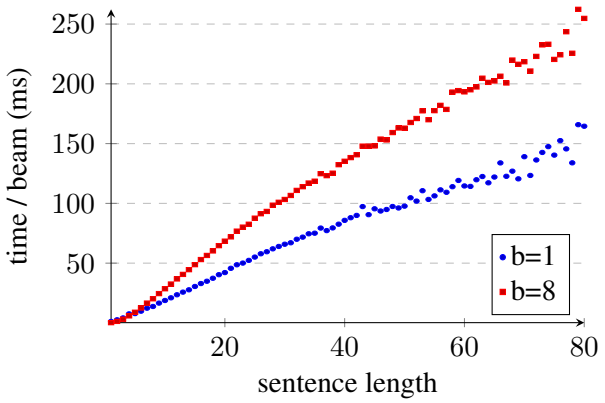

Figure 8: Parsing time w.r.t. sentence length (train)

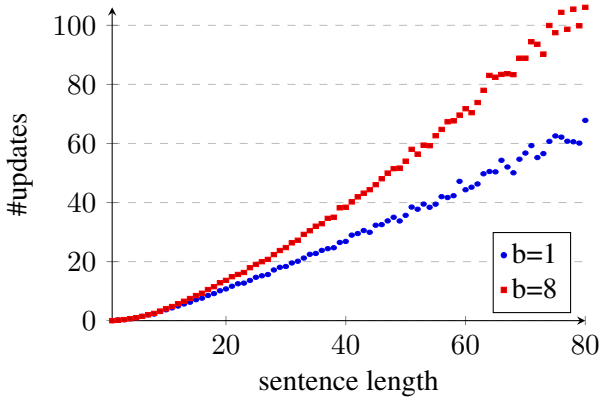

Figure 9: Number of updates w.r.t. sentence length (train)

shift-reduce parser with some preliminary support for training on ambiguous lattices. Although developed and tuned in less than a month, the participation of this very young system to the SPMRL 2013 shared task has shown its potential, even if far from the results of the best participants. As far as we know, DYALOG-SR is also the first system to show that shift-parsing techniques can be applied on ambiguous lattices, with almost no accuracy loss and with only minimal modifications (but large beams).

Several options are currently under consideration for improving the performances of DYALOG-SR. The first one is the (relatively straightforward) evolution of the parsing strategy for handling directly non-projective dependency trees, through the addition of some kind of SWAP transition (Nivre, 2009). Our preliminary experiments have shown the importance of larger beam sizes to cover the increased level of ambiguity due to lattices. However, it seems possible to adjust locally the beam size in function of the topology of the lattice, for improved accuracy and faster parsing. It also seems necessary to explore feature filtering, possibly using a tool like MALTOPTIMIZER (Ballesteros and Nivre, 2012), to determine the most discriminating ones.

The current implementation scales correctly w.r.t. sentence length and, to a lesser extent, beam size. Nevertheless, for efficiency reasons, we plan to implement a simple C module for beam management to avoid the manipulation in DYALOG of sorted lists. Interestingly, such a module, plus the already implemented model manager, should also be usable to speed up the disambiguation process of DYALOG-based TAG parser FRMG (de La Clergerie, 2005a). Actually, these components could be integrated in a slow but on-going effort to add first-class probabilities (or weights) in DYALOG, following the ideas of (Eisner and Filardo, 2011) or (Sato, 2008).

Clearly, DYALOG-SR is still at beta stage. However, for interested people, the sources are freely available[9], to be packaged in a near future.

Acknowledgements

We would like to thank the organizers of the SPMRL 2013 shared task and the providers of the datasets for the 9 languages of the task.

References

Anne Abeillé, Lionel Clément, and François Toussenel. 2003. Building a treebank for French. In Anne Abeillé, editor, *Treebanks*. Kluwer, Dordrecht.

Itziar Aduriz, M. J. Aranzabe, J. M. Arriola, A. Atutxa, A. Díaz de Ilarraza, A. Garmendia, and M. Oronoz. 2003. Construction of a Basque dependency treebank. In *TLT-03*, pages 201–204.

Miguel Ballesteros and Joakim Nivre. 2012. 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.

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The TIGER treebank. In Erhard Hinrichs and Kiril Simov, editors, *Proceedings of the First Workshop on Treebanks*

[9] via Subversion on INRIA GForge at `https://gforge.inria.fr/scm/viewvc.php/dyalog-sr/trunk/?root=dyalog`

and Linguistic Theories (TLT 2002), pages 24–41, Sozopol, Bulgaria.

Key-Sun Choi, Young S Han, Young G Han, and Oh W Kwon. 1994. KAIST tree bank project for Korean: Present and future development. In *Proceedings of the International Workshop on Sharable Natural Language Resources*, pages 7–14. Citeseer.

Jinho D. Choi. 2013. Preparing Korean Data for the Shared Task on Parsing Morphologically Rich Languages. *ArXiv e-prints*, September.

Dóra Csendes, Janós Csirik, Tibor Gyimóthy, and András Kocsor. 2005. The Szeged treebank. In Václav Matoušek, Pavel Mautner, and Tomáš Pavelka, editors, *Text, Speech and Dialogue: Proceedings of TSD 2005*. Springer.

Harold Charles Daume. 2006. *Practical structured learning techniques for natural language processing*. Ph.D. thesis, University of Southern California.

Éric de La Clergerie. 2005a. From metagrammars to factorized TAG/TIG parsers. In *Proceedings of IWPT'05 (poster)*, pages 190–191, Vancouver, Canada.

Éric de La Clergerie. 2005b. DyALog: a tabular logic programming based environment for NLP. In *Proceedings of 2nd International Workshop on Constraint Solving and Language Processing (CSLP'05)*, Barcelone, Espagne, October.

Jason Eisner and Nathaniel W. Filardo. 2011. Dyna: Extending Datalog for modern AI. In Tim Furche, Georg Gottlob, Giovanni Grasso, Oege de Moor, and Andrew Sellers, editors, *Datalog 2.0*, Lecture Notes in Computer Science. Springer. 40 pages.

Yoav Goldberg, Kai Zhao, and Liang Huang. 2013. Efficient implementation of beam-search incremental parsers. In *Proc. of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, Sophia, Bulgaria, August.

Yoav Goldberg. 2011. *Automatic syntactic processing of Modern Hebrew*. Ph.D. thesis, Ben Gurion University of the Negev.

Nizar Habash and Ryan Roth. 2009. Catib: The Columbia Arabic Treebank. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pages 221–224, Suntec, Singapore, August. Association for Computational Linguistics.

Nizar Habash, Reem Faraj, and Ryan Roth. 2009. Syntactic Annotation in the Columbia Arabic Treebank. In *Proceedings of MEDAR International Conference on Arabic Language Resources and Tools*, Cairo, Egypt.

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*, pages 1077–1086. Association for Computational Linguistics.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of HLT-NAACL 2012*, pages 142–151.

Mohamed Maamouri, Ann Bies, Tim Buckwalter, and Wigdan Mekki. 2004. The Penn Arabic Treebank: Building a Large-Scale Annotated Arabic Corpus. In *NEMLAR Conference on Arabic Language Resources and Tools*.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 99–106.

Joakim Nivre, Jens Nilsson, and Johan Hall. 2006. Talbanken05: A Swedish treebank with phrase structure and dependency annotation. In *Proceedings of LREC*, pages 1392–1395, Genoa, Italy.

Joakim Nivre. 2009. 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: Volume 1-Volume 1*, pages 351–359.

Benoît Sagot, Lionel Clément, Éric de La Clergerie, and Pierre Boullier. 2006. The Le*fff* 2 syntactic lexicon for French: architecture, acquisition, use. In *Proceedings of the 5th Language Resources and Evaluation Conference (LREC'06)*, Genova, Italie.

Taisuke Sato. 2008. A glimpse of symbolic-statistical modeling by PRISM. *J. Intell. Inf. Syst.*, 31(2):161–176.

Djamé Seddah, Reut Tsarfaty, Sandra Kübler, Marie Candito, Jinho Choi, Richárd Farkas, Jennifer Foster, Iakes Goenaga, Koldo Gojenola, Yoav Goldberg, Spence Green, Nizar Habash, Marco Kuhlmann, Wolfgang Maier, Joakim Nivre, Adam Przepiorkowski, Ryan Roth, Wolfgang Seeker, Yannick Versley, Veronika Vincze, Marcin Woliński, Alina Wróblewska, and Éric Villemonte de la Clergerie. 2013. Overview of the SPMRL 2013 shared task: A cross-framework evaluation of parsing morphologically rich languages. In *Proceedings of the 4th Workshop on Statistical Parsing of Morphologically Rich Languages: Shared Task*, Seattle, WA.

Wolfgang Seeker and Jonas Kuhn. 2012. Making Ellipses Explicit in Dependency Conversion for a German Treebank. In *Proceedings of the 8th International Conference on Language Resources and Evaluation*, pages 3132–3139, Istanbul, Turkey. European Language Resources Association (ELRA).

Khalil Sima'an, Alon Itai, Yoad Winter, Alon Altman, and Noa Nativ. 2001. Building a Tree-Bank for Modern Hebrew Text. In *Traitement Automatique des Langues*.

Marek Świdziński and Marcin Woliński. 2010. Towards a bank of constituent parse trees for Polish. In *Text, Speech and Dialogue: 13th International Conference (TSD)*, Lecture Notes in Artificial Intelligence, pages 197—204, Brno, Czech Republic. Springer.

Reut Tsarfaty. 2013. *A Unified Morpho-Syntactic Scheme of Stanford Dependencies*. Proceedings of ACL.

Veronika Vincze, Dóra Szauter, Attila Almási, György Móra, Zoltán Alexin, and János Csirik. 2010. Hungarian dependency treebank. In *LREC*.