Fast and Accurate Non-Projective Dependency Tree Linearization

Xiang Yu, Simon Tannert, Ngoc Thang Vu, Jonas Kuhn
Institut für Maschinelle Sprachverarbeitung
University of Stuttgart, Germany
firstname.lastname@ims.uni-stuttgart.de

Abstract

We propose a graph-based method to tackle the dependency tree linearization task. We formulate the task as a Traveling Salesman Problem (TSP), and use a biaffine attention model to calculate the edge costs. We facilitate the decoding by solving the TSP for each subtree and combining the solution into a projective tree. We then design a transition system as post-processing, inspired by non-projective transition-based parsing, to obtain non-projective sentences. Our proposed method outperforms the state-of-the-art linearizer while being 10 times faster in training and decoding.

1 Introduction

Surface realization is the task of generating a sentence from a syntactic or semantic representation. In several shared tasks (Belz et al., 2011; Mille et al., 2018, 2019), the input representations are unordered dependency trees. The state-of-the-art system (Yu et al., 2019a) in the Surface Realization Shared Task 2019 (SR'19) takes a pipeline approach, where the first step is linearization, namely ordering the tokens in the dependency tree. They use a Tree-LSTM to encode each token with awareness of the whole tree, then apply the divide-and-conquer strategy to split the full tree into subtrees and find the optimal order for each subtree using beam search. Finally, the linearized subtrees are combined into a full projective tree. The general strategy is adapted from Bohnet et al. (2010).

In this work, we tackle linearization decoding in a different way, by casting it as a Traveling Salesman Problem (TSP). Knight (1999) first formulated the word ordering of the target language in word-based machine translation as a TSP, where the words are the nodes to traverse, and the log probabilities of the bigrams are the edge costs.

Several works have followed this formulation. Among others, Zaslavskiy et al. (2009) formulate the word ordering in phrase-based machine translation as a TSP, and show that it achieves better performance and speed than beam search decoding with the same bigram language model. Horvat and Byrne (2014) explore higher-order n-gram language models for TSP-based word ordering, which transforms into a much larger TSP graph. All of the aforementioned works operate on a bag of words *without syntax*, which is a TSP graph of non-trivial size with little information about the internal structure. Much effort has been put into incorporating more powerful decoding algorithms such as Integer Programming (Germann et al., 2001) and Dynamic Programming (Tillmann and Ney, 2003).

Our work differs from the previous work on TSP-based word ordering in several aspects. (1) Linearization is a special case of word ordering *with syntax*, where we can use a tree-structured encoder to provide better representation of the tokens. (2) We adopt the divide-and-conquer strategy to break down the full tree into subtrees and order each subtree separately, which is faster and more reliable with an approximate decoder. (3) We apply deep biaffine attention (Dozat and Manning, 2016), which has yielded great improvements in dependency parsing, and reinterpret it as a bigram language model to compute edge costs for the TSP.

In this paper, we solve the dependency tree linearization task as a TSP. With the help of Tree-LSTM to encode the tree and biaffine attention as a bigram language model, we can use a greedy TSP solver to linearize the tree effectively. Furthermore, the divide-and-conquer strategy greatly reduces the search space but introduces the projectivity restriction, which we remedy with a transition-based reordering system. As a result, the proposed linearizer outperforms the previous state-of-the-art model both in quality and speed.

Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 1451–1462

2 Graph-based Linearization

2.1 System Overview

We follow the idea in Knight (1999) to treat linearization as a TSP. Under the TSP formulation, we need to calculate the cost from every node i to every other node j, which can be interpreted as the log likelihood of the bigram (i, j). We use the biaffine attention model (Dozat and Manning, 2016) to obtain the costs, and use an off-the-shelf TSP solver, OR-Tools[1], to decode the TSP.

To facilitate the approximate decoding of this NP-hard problem, we follow the divide-and-conquer strategy in Bohnet et al. (2010) of splitting the tree into subtrees, and decoding each subtree separately. There are pros and cons of this approach: on the one hand, the search space is much smaller so that a greedy TSP solver can find good solutions in reasonable time; on the other hand, it restricts the output to be projective, i.e., non-projective sentences can never be produced.

To remedy the projectivity restriction, we introduce a post-processing step using a simple transition system with only two transitions, *swap* and *shift*, to sort the linearized projective trees into potentially non-projective ones.

This system is an extention of our previous work (Yu et al., 2019a). We use the same encoder and hyperparameters (see Appendix A) and only experiment with the decoders. The code is available at the first author's web page.[2]

As an overview, Figure 1 illustrates our pipeline for the linearization task, with an unordered dependency tree as input, and a linearized sentence as output, which is potentially non-projective, i.e., with crossing dependency arcs.

To solve the task, we (1) divide the tree into subtrees, (2) linearize each subtree by solving a TSP, (3) combine the linearized subtrees into an projective tree, and (4) use the swap system to obtain a non-projective tree.

2.2 TSP Solver

To formulate the linearization task as a TSP, we use a node to represent each token in the tree, and an extra node with index 0 as both the origin and destination, which is interpreted as the boundaries

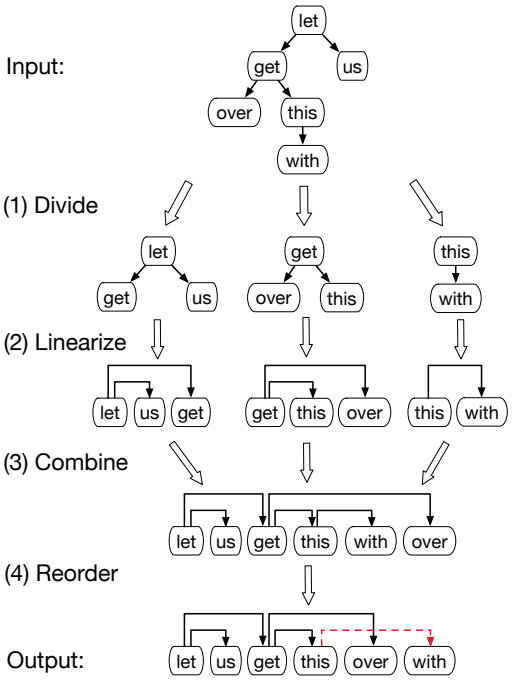
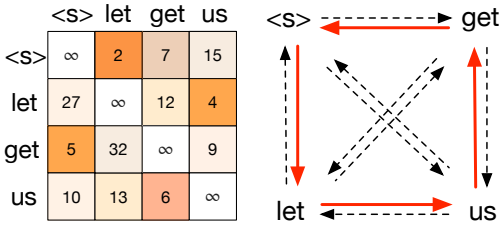


Figure 1: An overview of the linearization system.



(a) Cost matrix (b) TSP graph

Figure 2: The graph and matrix views of a TSP.

in the output sequence. Figure 2 demonstrates a decoded TSP graph from its edge cost matrix, where the output sequence is "<s> let us get <s>".

We use the routing solver of OR-Tools to solve the TSP given the edge costs. It is a generic optimization library, unlike the more specialized and optimized TSP solvers such as Concorde (Applegate et al., 2006), but it enables imposing extra word order constraints described in §2.6, and can be easily extended to other constraints.

Among all the first solution strategies provided in OR-Tools, we found `GLOBAL_CHEAPEST_ARC` to perform the best, which selects a valid edge with the lowest cost at every step until a full path is found. For the sake of efficiency, we use the greedy metaheuristic `GREEDY_DESCENT` to refine the first solutions, which converges to local optima in very short time. In practice, it works extremely well in combination with the greedy training described in §2.4.

[1] https://developers.google.com/optimization

[2] https://www.ims.uni-stuttgart.de/en/institute/team/Yu-00010/

More advanced metaheuristics such as GUIDED_LOCAL_SEARCH (Voudouris and Tsang, 1999) could find better solutions, but also require much more decoding time, it is thus less practical for real-time generation tasks. We do not use it in the default setting, but include it in the analysis to demonstrate the effectiveness of the training.

2.3 Scoring Model

We use the biaffine attention model (Dozat and Manning, 2016) to calculate the TSP edge costs. First, we obtain the representation for each token by concatenating the embeddings of the features, then encode the tree information with a bidirectional Tree-LSTM, as described in Yu et al. (2019b):

$$\mathbf{v}_i^\circ = \mathbf{v}_i^{(lem)} \oplus \mathbf{v}_i^{(pos)} \oplus \mathbf{v}_i^{(dep)} \oplus \mathbf{v}_i^{(mor)} \quad (1)$$

$$\mathbf{x}_i = \text{Tree-LSTM}(\mathbf{v}_0^\circ ... \mathbf{v}_n^\circ)_i \quad (2)$$

The parameters of the decoder consist of two Multi-Layer Perceptrons ($\text{MLP}^{(fr)}$ and $\text{MLP}^{(to)}$) and a biaffine matrix ($\mathbf{W}$). We use the MLPs to obtain different views of the token representation as the first and second token in the bigram:

$$\mathbf{h}_i^{(fr)} = \text{MLP}^{(fr)}(\mathbf{x}_i) \quad (3)$$

$$\mathbf{h}_i^{(to)} = \text{MLP}^{(to)}(\mathbf{x}_i) \quad (4)$$

We then apply the biaffine transformation on the vectors of the first word $\mathbf{h}_i^{(fr)}$ and the second word $\mathbf{h}_j^{(to)}$ to compute the score $s_{i,j}$ of each bigram (i, j), where $\mathbf{W}$ is the weight matrix of size $(k + 1) \times (k + 1)$, and k is the size of $\mathbf{h}_i^{fr}$ and $\mathbf{h}_j^{to}$:

$$s_{i,j} = (\mathbf{h}_i^{(fr)} \oplus 1) \times \mathbf{W} \times (\mathbf{h}_j^{(to)} \oplus 1)^\top \quad (5)$$

In the actual computation, we parallelize Equation 5, where $\mathbf{S}$ is the output score matrix of size $n \times n$, and n is the number of tokens:

$$\mathbf{S} = (\mathbf{H}^{(fr)} \oplus \mathbf{1}) \times \mathbf{W} \times (\mathbf{H}^{(to)} \oplus \mathbf{1})^\top \quad (6)$$

Finally, we turn the score matrix into a non-negative cost matrix for the TSP solver:

$$\mathbf{C} = \max(\mathbf{S}) - \mathbf{S} \quad (7)$$

Our model is inspired by the biaffine dependency parser of Dozat and Manning (2016), but stands in contrast in many aspects. They use a bidirectional LSTM to encode the sequential information of the tokens, and the biaffine attention itself does not model the sequence. Each cell $s_{i,j}$ in their output matrix $\mathbf{S}$ is interpreted as the score of a dependency arc (i, j). They use a Maximal Spanning Tree algorithm to obtain a tree that maximizes the total score of the arcs in the tree.

In the case of linearization, our input and output are the opposite to theirs. The input has no sequential but syntactic information, encoded by the bidirectional Tree-LSTM. Each cell $s_{i,j}$ in the output matrix $\mathbf{S}$ is interpreted as the score of the bigram (i, j). We use a TSP solver to obtain a traversal of the tokens by minimizing the total edge costs, i.e., maximizing the total bigram scores.

2.4 Training Objective

We use a greedy training objective to train the biaffine scoring model, namely we enforce the score of each bigram (i, j) in the correct sequence z to be higher than any other bigrams in the same row or in the same column in the matrix by a margin:

$$L = \sum_{(i,j)\in z} \Big(\sum_{j' \neq j} \max(0, 1 + s_{i,j'} - s_{i,j}) + \sum_{i' \neq i} \max(0, 1 + s_{i',j} - s_{i,j}) \Big) \quad (8)$$

This objective aims to maximizing the score of each correct bigram (i, j) in both directions, essentially $\log P(j|i)$ and $\log P(i|j)$, where the cells in the same row corresponds to all possible tokens following i, and the cells in the column corresponds to all possible tokens preceding j.

The objective is greedy in the sense that it updates more than "necessary" to decode the correct path. We contrast it to the structured loss in most graph-based dependency parsers (McDonald et al., 2005; Kiperwasser and Goldberg, 2016), which updates the scores of the correct path z against the highest scoring incorrect path z':

$$L' = \max(0, 1 + \max_{z' \neq z} \sum_{(i',j')\in z'} s_{i',j'} - \sum_{(i,j)\in z} s_{i,j}) \quad (9)$$

The greedy objective for the TSP has two main advantages: (1) it does not require decoding during training, which saves training time; (2) it pushes the scores of each correct bigram to be the highest in the row *and* the column, which facilitates the greedy solver (GLOBAL_CHEAPEST_ARC) to find a good initial solution. In fact, if the objective

reaches 0, the greedy solver is guaranteed to find the optimal solution, since at each step, the cheapest arc is always a correct bigram instead of any other bigram in the same row or column.

2.5 Generating Non-Projective Trees

If we directly linearize the full tree, the output is naturally unrestricted, i.e., possibly non-projective. However, when we linearize each subtree separately in order to reduce the search space, as in the proposed method, the reconstructed output is restricted to be projective (Bohnet et al., 2010).

To relax the projectivity restriction, we design a transition system to reorder projective trees into non-projective trees as a post-processing step, inspired by Nivre (2009) but working in the opposite way. It is essentially a reduced version of their transition system, removing the attachment transitions and keeping only *swap* and *shift*.

In the transition system (as shown in Table 1), a configuration consists of a stack σ, which is initially empty, and a buffer β, which initially holds all input tokens. The *shift* transition moves the front of the buffer to the top of the stack, and the *swap* transition moves the top of the stack back to the second place in the buffer. When all tokens are moved from the buffer to the stack, the procedure terminates. To prevent the model predicting infinite *shift-swap* loops, we only allow *swap* if the initial index of the top of the stack is smaller than the front of the buffer. The worst-case complexity of the sorting is quadratic to the number of tokens, however, since trees in natural language mostly only contain very few non-projective arcs, the transition system works in expected linear time, as shown in Nivre (2009).

We then implement a model to predict the transitions given the configurations. We use two LSTMs to dynamically encode the stack from left to right (LSTM_σ) and the buffer from right to left (LSTM_β). We then concatenate the two outputs and use a MLP to predict the next transition.

When a *shift* is performed, we update LSTM_σ state with the vector of the shifted token as the new stack representation, and the new buffer representation is the LSTM_β output of the new front token; when a *swap* is performed, the new stack representation is the LSTM_σ output of the new top token, and the new buffer representation is recalculated by feeding the now second and first token in the buffer to the LSTM_β state of the third token.

Transition	Before		After
shift	$(\sigma,\ [i\|\beta])$	$\rightarrow$	$([\sigma\|i],\ \beta)$
swap	$([\sigma\|i],\ [j\|\beta])$	$\rightarrow$	$(\sigma,\ [j\|i\|\beta])$

Table 1: The *shift-swap* transition system.

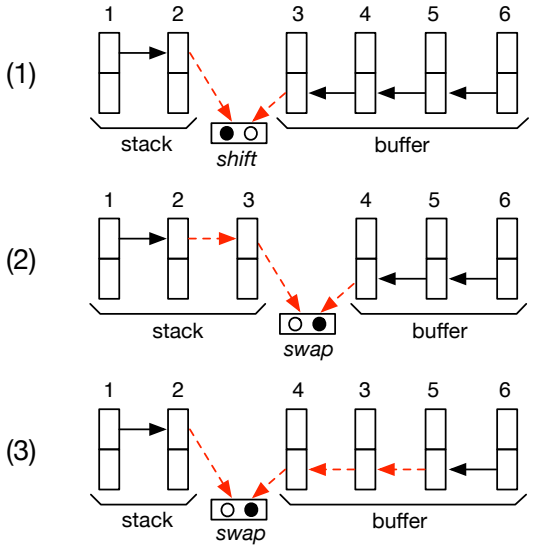


Figure 3: An illustration of the swap model. (1) shows a configuration [1 2 | 3 4 5 6], where the stack contains 1 and 2, and the buffer contains 3, 4, 5 and 6, and the model predicts *shift* to move 3 from the buffer to the stack; (2) is the resulted configuration, and the model predicts *swap*, which moves 3 back to the buffer behind 4, as shown in (3). The solid black arrows represent the computation already done in the previous steps, and the dashed red arrows represent the new computation needed for the current step.

Figure 3 illustrates the model under the transition system, where the arrows to the right represent LSTM_σ, the arrows to the left represent LSTM_β, and the arrows between the stack and the buffer represent the MLP. After each transition, little computation is needed to represent the new stack and buffer, marked with the red dashed line. The example illustrates the steps to modify the configuration [1 2 | 3 4 5 6] into [1 2 | 4 3 5 6].

Note that the transition system is sound and complete, which means there is always a sequence of transitions to sort any sequence into any reordering. In other words, the transition system on its own could also linearize the tree by taking a random permutation as input. However, due to the noisy input order, it is very difficult for the LSTM model to learn good representations for the stack and buffer and predict correct transitions (cf. Vinyals et al. (2015) for the discussion on encoding a set with an LSTM). In contrast, when we only use this sys-

tem to reorder a linearized projective tree as post-processing, where input sequence is meaningful and consistent, it is much easier to learn.

Using the swap system as a post-processing step stands in contrast to Bohnet et al. (2012), where they pre-process the tree by lifting the arcs so that the correct word order could form a projective tree. These two approaches draw inspiration from the non-projective parsing in Nivre (2009) and the pseudo-projective parsing in Nivre and Nilsson (2005) respectively. We argue that our post-processing approach is more convenient since there is no need to change the syntactic annotation in the original tree, and it is much easier to evaluate the effectiveness of the sorting model.

2.6 Relative Word Order Constraints

In the SR'19 dataset, some relative word order information is given, which indicates e.g. the order of the conjuncts in the coordination. Since the order in a coordination is generally arbitrary (at least syntactically), it will thus introduce randomness in the single reference evaluation. We believe that using such information leads to more accurate evaluation, and therefore by default always use these constraints in the comparison.

The constraints does not specify direct adjacency, but only general precedence relations. For example, to order the nodes $\{1, 2, 3, 4, 5\}$ with the constraint $2 \prec 3 \prec 1$ and $4 \prec 5$, a valid sequence could be $[2, 4, 3, 5, 1]$, while $[4, 5, 2, 1, 3]$ is invalid.

To incorporate such constraints in the solver, we introduce an additional variable associated with each node in the routing problem, where the value is incremented by 1 after each step. In other words, if a node is visited in the n-th step, then the associated variable will have the value n. Then we add inequality constraints about those variables that are specified in the word order information into the routing problem and let the solver find the path that satisfies the constraints.

In practice, the solver can always find a solution to linearize the subtrees with the constraints. However, it sometimes cannot find any solution to directly linearize the full tree within the time limit (1-10% of the cases depending on the treebank), because there are more nodes and more constraints in the full tree. In this case, we simply remove the constraints and rerun the solver.

3 Experiments

3.1 Data and Baselines

We use the datasets from the Surface Realization 2019 Shared Task (Mille et al., 2019) in our experiments, which includes 11 languages in 20 treebanks from the Universal Dependencies (Nivre et al., 2016). We experiment on the shallow track, i.e., all tokens in the output are present in the input tree. We only report the BLEU score (Papineni et al., 2002) as the evaluation metric, since we mostly evaluate on the lemma level, where the metrics involving word forms are irrelevant.

As baselines for the final evaluation, we use several available linearizers by Bohnet et al. (2010) (B10), Puduppully et al. (2016) (P16) and Yu et al. (2019a) (Y19). B10, P16 and our linearizer all use the same inflection and contraction models, trained with the same hyperparameters as in Y19, and we compare to the reported shared task results of Y19.

3.2 Main Results

Table 2 shows the performance of different linearizers, where `beam` is the baseline beam-search linearizer as in Yu et al. (2019b) with default hyperparameters, `full` is the TSP decoder on the full tree level, `sub` is the TSP decoder on the subtree level, and `+swap` is `sub` post-processed with reordering. We test the decoders under two conditions: without word order constraints (`-constraints`) and with word order constraints (`+constraints`). Columns 2-9 show the BLEU scores on lemmata on the development set, and in the last 4 columns are the BLEU scores on inflected and contracted word forms on the test sets with the official evaluation script of SR'19.

While both only generating projective sentences, the `sub` decoder outperforms the baseline `beam` decoder by 0.6 BLEU points without word order constraints and 0.3 BLEU points with constraints. Note that the beam search decoder uses an LSTM to score the sequences, which is essentially an unlimited language model, while the TSP decoders only uses a bigram language model.

While comparing the two TSP decoders, `sub` performs on average higher than `full`, while `full` performs better on treebanks with more non-projective sentences, since it is not restricted. Without word order constraints, `full` even slightly outperforms `sub`. The reason that `full` performs relatively worse with constraints is that it sometimes has to remove the constraints to find a solution.

	%NP	-constraints beam	full	sub	+swap	+constraints beam	full	sub	+swap	Final Test B10	P16	Y19	Ours
ar_padt	0.48	85.09	85.00	**85.67**	**85.67**	**86.74**	85.54	**86.74**	**86.74**	56.62	56.17	64.90	**67.02**
en_ewt	0.62	85.19	85.30	**85.99**	**85.99**	**88.38**	87.40	88.01	88.03	72.97	74.53	82.98	**84.08**
en_gum	1.00	84.75	85.20	85.86	**86.00**	86.96	86.53	87.43	**87.55**	69.94	70.88	83.84	**84.72**
en_lines	3.81	79.23	81.60	80.04	**81.63**	81.71	82.92	81.97	**83.51**	63.15	67.67	81.00	**81.55**
en_partut	0.34	87.05	**87.90**	86.73	86.73	88.13	87.03	**88.77**	**88.77**	80.64	70.97	**87.25**	85.52
es_ancora	0.88	83.68	**85.00**	84.21	84.70	84.88	85.29	85.55	**86.09**	80.80	69.73	83.70	**85.34**
es_gsd	0.51	83.68	**84.70**	84.26	84.31	85.70	85.62	86.29	**86.34**	79.18	70.34	**82.98**	82.78
fr_gsd	0.61	87.13	**88.00**	87.58	87.60	89.41	88.88	89.68	**89.77**	79.34	72.25	**83.95**	83.34
fr_partut	0.46	87.56	88.40	**90.38**	**90.38**	90.07	89.76	**90.58**	**90.58**	75.13	64.20	**83.38**	83.21
fr_sequoia	0.27	87.20	85.50	87.13	**87.29**	**89.74**	86.85	89.44	89.60	77.48	62.67	**84.52**	83.81
hi_hdtb	1.20	83.14	85.10	83.82	**85.37**	85.04	86.32	85.50	**87.06**	77.89	74.70	80.56	**82.54**
id_gsd	0.57	81.65	81.80	81.90	**82.06**	85.56	82.45	86.30	**86.44**	77.90	76.51	85.34	**85.57**
ja_gsd	0.14	**90.66**	90.10	90.50	90.50	**92.83**	91.65	**92.83**	**92.83**	83.67	81.21	87.69	**87.87**
ko_gsd	3.10	76.29	76.60	75.73	**77.21**	79.26	79.39	79.47	**80.87**	61.76	65.89	74.19	**75.12**
ko_kaist	3.59	79.24	**84.00**	80.52	83.18	80.29	**84.50**	80.84	83.53	63.48	71.41	73.93	**77.50**
pt_bosque	2.95	82.57	**84.40**	83.04	84.02	83.97	**85.46**	84.40	85.19	75.41	67.91	77.75	**79.15**
pt_gsd	0.44	87.45	**88.40**	87.91	88.15	88.96	89.12	89.83	**90.06**	74.44	68.11	75.93	**77.00**
ru_gsd	0.84	74.50	73.80	74.99	**75.06**	**79.07**	75.37	78.86	78.87	63.32	62.93	71.23	**71.27**
ru_syntagrus	1.14	77.95	78.90	79.23	**79.39**	81.00	80.77	81.59	**81.71**	74.28	71.50	76.94	**78.39**
zh_gsd	0.06	81.83	80.20	**82.25**	**82.25**	**83.29**	79.65	82.93	82.93	77.88	70.55	83.85	**84.76**
AVG	1.15	83.29	84.04	83.89	**84.34**	85.55	85.03	85.85	**86.32**	73.26	69.51	80.30	**81.03**

Table 2: Percentage of non-projective arcs (column 1); BLEU scores on lemmata on the development set for different linearization decoders, with linear order constraints (column 2-5) and without linear order constraints (column 6-9); and BLEU scores on inflected words on the test set compared with several baseline systems (column 10-13), where B10 denotes Bohnet et al. (2010), P16 denotes Puduppully et al. (2016), and Y19 denotes Yu et al. (2019a). The best result in each group is marked with bold face.

The `sub+swap` decoder eliminates the projectivity restriction, closing the performance gap to `full` for non-projective treebanks, and it does not hurt the performance on the projective treebanks.

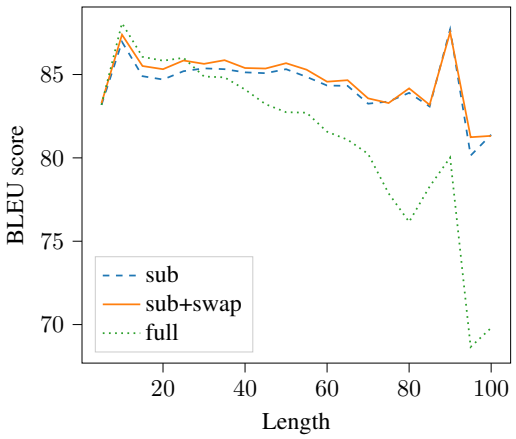
In the last four columns we compare our `sub+swap` linearizer on the test set for the full pipeline with three external baselines, including the best system in the SR'19 shared task (Y19). Our system outperforms B10 and P16 by a large margin of 7 and 11 BLEU points. Note that their off-the-shelf systems are not designed to use word order constraints and morphological tags, which would account for a difference of about 3 points (see the effect of constraints in Table 2 and feature ablation in §3.7). Under the same condition, our system outperforms Y19 on most of the treebanks and on average by 0.7, because of (1) a better projective decoder and (2) the non-projective post-processing step. Furthermore, our system is much faster than Y19, see the comparison in §3.8.

3.3 Error Analysis

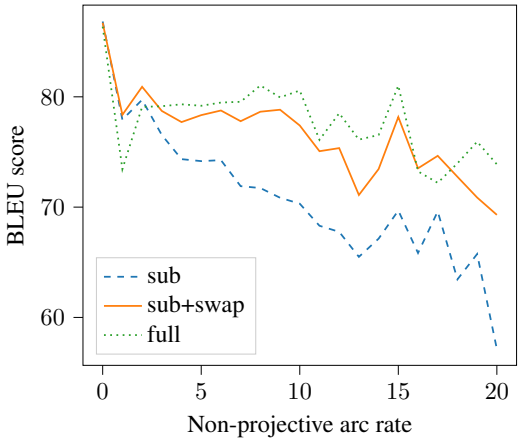
To illustrate the characteristics of different TSP decoders, we analyze their performance on sentences with different lengths and percentages of non-projective arcs.

Figure 4a shows the BLEU score of different TSP decoders with respect to the sentence length, averaged over all sentences in the development sets. The `sub` model performs quite stably across the sentences with different lengths, while the `full` model performs much worse on longer sentences.[3] This confirms our hypothesis that the divide-and-conquer strategy of the subtree decoder can reduce search errors for large TSP problems. Post-processing with the swap system (`tsp+swap`) consistently improvements `tsp` across all sentence lengths.

[3]Note that the very short sentences have even lower BLEU score, this is caused by the smoothing function in the BLEU evaluation, which gives a low score even for exact match.



(a) BLEU vs. Sentence lengths.



(b) BLEU vs. non-projective arc rate.

Figure 4: BLEU score with respect to the sentence length and the percentage of non-projective arcs.

Figure 4b shows the BLEU score with respect to the percentage of non-projective arcs in the gold tree, averaged over all sentences in the development sets. Clearly, `sub` performs lower than `full` for sentences with more non-projective arcs due to the projectivity restriction, while the overall BLEU score of `sub` is higher, since 99% of the arcs and 90% of the sentences are projective. With the help of the swap system, `sub+swap` closes the gap to `full` on the non-projective sentences.

In sum, the `sub+swap` model shows clear advantages over the other models since it is less prone to search error due to the reduced TSP size and free from the projectivity restriction, it is thus the best of both worlds.

3.4 Training Objective

As described in §2.4, we use a greedy training objective to train the biaffine model, namely we calculate a hinge loss of the correct bigram against all other bigrams in the same row and in the same column. This is in contrast to the structured loss, which is calculated between the gold sequence and the predicted sequence.

	row+col	row	col	path
sub	85.85	85.56	85.60	85.13
full	85.03	84.36	84.38	80.96

Table 3: BLEU scores of different decoders with different training objectives.

	BLEU			Δ_{BLEU}		
	gold	pred	rand	gold	pred	rand
gold	98.50	86.32	2.63	+0.67	+0.47	+0.13
pred	97.52	85.78	3.61	-0.32	-0.07	+1.11
rand	87.83	81.97	71.36	-10.01	-3.88	+68.86

Table 4: The change in BLEU scores after applying the transition-based sorting models that are trained (row) and tested (column) under different conditions.

This contrast is similar to the two different training objective in Dozat and Manning (2016) against Kiperwasser and Goldberg (2016) for graph-based dependency parsing. We experiment with the structured loss, following Kiperwasser and Goldberg (2016), where we also apply loss augmented inference (Taskar et al., 2005), i.e., adding a constant for all the bigrams that are not in the gold sequence.

We also experiment with only updating against the row or the column, which could be thought of as the bigram language model only in one direction, while updating against both is training a bidirectional language model.

Table 3 shows the results, where we train the `sub` and `full` models with different objectives: `row+col` is the default one, `row` and `col` only update against the row or the column, and `path` updates the gold path against the predicted path.

The results are clear: for both `sub` and `full` models, training on both directions is better than training on one direction, and the greedy objective is better than the structured objective. The gap between the bidirectional greedy objective and others is larger in `full` than in `sub`, since `full` solves a larger TSP, where the greedy training is even more important for effective greedy decoding.

3.5 Transition-based Sorting

As discussed in §2.5, we use the transition system only for post-processing the linearized projective sentences, although the transition system itself is theoretically able to sort a random sequence. The question is whether the model is able to learn to handle the random input.

We experiment with different training and testing scenarios of the sorting models. They are trained in three scenarios, namely to sort (1) gold projective sentences into correct (potentially non-projective) sentences, noted as `gold`; (2) predicted projective sentences, where the sentences are obtained by 5-fold jackknifing on the training set using the `sub` model, noted as `pred`; and (3) random sequences, where the input is always shuffled during training, noted as `rand`. The models are then applied to sort (1) gold projective sentences (`gold`); (2) predicted projective sentences from the `sub` model (`pred`); and (3) random permutation of the tokens (`rand`). In the main experiment, the way we use the transition system corresponds to the `gold-pred` scenario.

Table 4 shows the BLEU scores on the development set averaged over all treebanks. We also show the change of BLEU scores from the input to the output (Δ_{BLEU}) in different scenarios.

First, the `gold` model improves the input in all scenarios, especially the `gold-pred` scenario used in the main experiment brings 0.47 BLEU points improvement. Interestingly, the `pred` model from jackknifing does not improve the performance, while usually training on the data with erroneous prediction should prevent overfitting to the gold data. We conjecture the reason could be that the model is overfitting to fixing the particular errors in the predicted training data instead of learning to produce non-projective sentences.

Purely using the transition system for linearization (`rand-rand`) works to some extent, but performs lower than the baseline by a large gap for several reasons. First, it imposes an arbitrary order in the input which is a suboptimal way to represent a bag of word. Second, learning to sort random permutation requires a lot more training instances to generalize. Finally, it takes on average $\mathcal{O}(n^2)$ steps, which also increases the chance of error propagation. In contrast, sorting a projective tree does not have any of these disadvantages.

Generally, when the training and testing scenarios are not aligned, the performance is always worse due to the mismatched bias of transitions. For example, `gold-rand` barely changes the random input since it mostly predicts *shift*, and `rand-gold` predicts *swap* too often such that the outcome is even worse than the input sentence.

	beam	sub	full	swap
+tree	85.55	85.85	85.03	71.36
-tree	78.31	74.17	35.82	19.74
Δ	-7.24	-11.68	-49.21	-51.62

Table 5: Comparing the decoders with and without the Tree-LSTM encoder.

3.6 Syntax Ablation

The success of the simple bigram language model and greedy TSP decoding relies heavily on the Tree-LSTM encoding. To demonstrate its importance, we remove the tree encoding for each linearizer, i.e., they only receive the token level features as the representation. We experiment with four linearizers: apart from `beam`, `sub` and `full` as in the main experiments, we also include the `swap` linearizer that is trained to sort *random* input sequences. The condition `+tree` is the default case, while in `-tree` we do not use the tree encoding. Note that in the latter case, `beam` and `sub` still use the tree information to split the tree into subtrees, while `full` and `swap` do not use the tree information in any way.

The results are shown in Table 5. Without the tree encoder, the performance drop in `sub` is larger than `beam`, which suggests that `sub` is more dependent on the good representation of the Tree-LSTM encoder, since its scoring function is essentially a bigram language model, which would be much less expressive than the LSTM in `beam` if syntax is absent. This result draws an interesting analogy to the fact that first-order graph-based dependency parsers (Kiperwasser and Goldberg, 2016; Dozat and Manning, 2016) also outperform the transition-based counterparts with a simpler scoring model but without error propagation.

The much larger drop in `full` and `swap` emphasizes the importance of the inductive bias introduced by the divide-and-conquer strategy, since natural languages are predominantly projective.

Generally, the syntax ablation experiment highlights the crucial difference between our work and the original idea by Knight (1999), namely we use *contextualized* bigrams in our TSP model, which

is much more expressive than the vanilla version. Consider the subtree with the words "this" and "with" in Figure 1, a vanilla bigram model would calculate a much higher score for "with this" than "this with", while a contextualized bigram model could be aware that it is part of a rather special syntactic construction in English.

3.7 Feature Ablation

To understand how much each feature contributes to the linearization task, we perform ablation experiments on the selection of features. In the default setting of our models, we use the lemma, UPOS, dependency relation, and morphological tags to encode each token. We experiment with turning off each feature for the `sub` linearizer, as well as only using one feature, and the results are in Table 6. The results suggest that the UPOS tags and morphological tags do not provide much additional information and could be dropped if simplicity is desired. In contrast, the lemmata and dependency relations are crucial to determine the word order, since the performance drops considerably without them.

	none	lemma	dep	upos	morph
without	85.85	83.12	82.49	85.56	85.58
with only	-	79.51	81.11	76.14	79.31

Table 6: Feature ablation experiments, where we test removing one feature (first row) and using only one feature (second row).

3.8 Performance vs. Speed

By default, we use a greedy TSP solver, which already yields satisfactory performance. We then make additional experiments with a more optimized metaheuristic (guided local search) to see if better performance can be gained in exchange for more decoding time. With the guided local search, we set the search limit to 1 second or 100 solutions for each subtree, and 10 seconds or 1000 solutions for the full tree. We also compare to the beam search linearizer with varying beam sizes from 1 to 64. The results are shown in Figure 5, where the decoding time is measured on a single CPU core.

Generally, all greedy TSP solvers outperform the Pareto front of the beam search decoders. The greedy solver performs almost as well as the optimized solver for the subtree TSP (85.85 vs. 85.91), while it performs clearly worse for the full tree

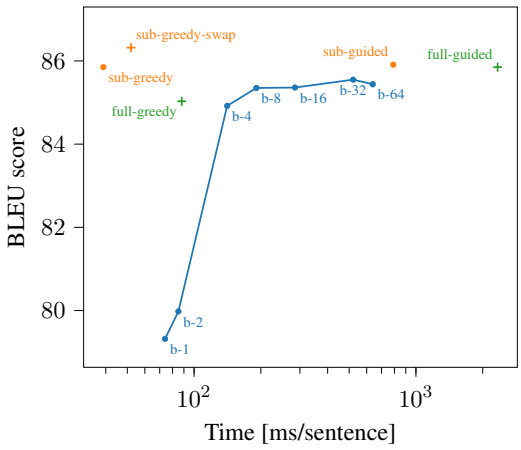


Figure 5: The speed (on a log scale) and BLEU score for different decoders, where the dots are projective decoders, and the crosses are non-projective decoders.

TSP (85.03 vs. 85.85). This contrast again demonstrates that the divide-and-conquer strategy indeed greatly simplifies the problem for the greedy solver. Post-processing with the swap system only slightly increase the decoding time (in total 50ms per sentence), but considerably improves the performance.

4 Conclusion

In this paper, we revisit the idea of treating word ordering as a TSP, but unlike the common bag-of-words scenario, the words have an underlying syntactic structure. We demonstrate that with the Tree-LSTM encoder, the biaffine scoring model, the divide-and-conquer strategy, and a transition-based sorting system, we can linearize a dependency tree with high speed and quality and without the projectivity restriction. We show with various ablation experiments that all of the components are crucial for the success of the TSP-based linearizer.

Our work emphasizes the importance of syntax in the word ordering task. We discussed many connections and similarities between linearization and parsing. We believe that quite generally, systems for solving one task can benefit from the other task's view on syntactic structure. One possibility to capitalize on these synergies is to explore data augmentation methods to select beneficial extra training data in an unsupervised fashion.

Acknowledgements

This work was in part supported by funding from the Ministry of Science, Research and the Arts of the State of Baden-Württemberg (MWK), within the CLARIN-D research project.

References

David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. 2006. Concorde TSP Solver.

Anja Belz, Mike White, Dominic Espinosa, Eric Kow, Deirdre Hogan, and Amanda Stent. 2011. The First Surface Realisation Shared Task: Overview and Evaluation Results. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 217–226, Nancy, France. Association for Computational Linguistics.

Bernd Bohnet, Anders Björkelund, Jonas Kuhn, Wolfgang Seeker, and Sina Zarriess. 2012. Generating Non-Projective Word Order in Statistical Linearization. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 928–939, Jeju Island, Korea. Association for Computational Linguistics.

Bernd Bohnet, Leo Wanner, Simon Mille, and Alicia Burga. 2010. Broad Coverage Multilingual Deep Sentence Generation with a Stochastic Multi-Level Realizer. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 98–106. Association for Computational Linguistics.

Timothy Dozat and Christopher D. Manning. 2016. Deep Biaffine Attention for Neural Dependency Parsing. *ArXiv*, abs/1611.01734.

Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. 2001. Fast Decoding and Optimal Decoding for Machine Translation. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, pages 228–235, Toulouse, France. Association for Computational Linguistics.

Matic Horvat and William Byrne. 2014. A Graph-Based Approach to String Regeneration. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 85–95, Gothenburg, Sweden. Association for Computational Linguistics.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations. *Transactions of the Association for Computational Linguistics*, 4:313–327.

Kevin Knight. 1999. Decoding Complexity in Word-Replacement Translation Models. *Computational Linguistics*, 25(4):607–615.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Simon Mille, Anja Belz, Bernd Bohnet, Yvette Graham, Emily Pitler, and Leo Wanner. 2018. The First Multilingual Surface Realisation Shared Task (SR'18): Overview and Evaluation Results. In *Proceedings of the First Workshop on Multilingual Surface Realisation*, pages 1–12, Melbourne, Australia. Association for Computational Linguistics.

Simon Mille, Anja Belz, Bernd Bohnet, Yvette Graham, and Leo Wanner. 2019. The Second Multilingual Surface Realisation Shared Task (SR'19): Overview and Evaluation Results. In *Proceedings of the 2nd Workshop on Multilingual Surface Realisation (MSR 2019)*, pages 1–17, Hong Kong, China. Association for Computational Linguistics.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980*.

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*, pages 351–359, Suntec, Singapore. Association for Computational Linguistics.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal Dependencies v1: A Multilingual Treebank Collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666, Portorož, Slovenia. European Language Resources Association (ELRA).

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

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Ratish Puduppully, Yue Zhang, and Manish Shrivastava. 2016. Transition-Based Syntactic Linearization with Lookahead Features. In *Proceedings of*

the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 488–493, San Diego, California. Association for Computational Linguistics.

Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. 2005. Learning structured prediction models: A large margin approach. In *Proceedings of the 22nd international conference on Machine learning*, pages 896–903. ACM.

Christoph Tillmann and Hermann Ney. 2003. Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation. *Computational Linguistics*, 29(1):97–133.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2015. Order Matters: Sequence to Sequence for Sets. *arXiv preprint arXiv:1511.06391*.

Christos Voudouris and Edward Tsang. 1999. Guided Local Search and its Application to the Traveling Salesman Problem. *European journal of operational research*, 113(2):469–499.

Xiang Yu, Agnieszka Falenska, Marina Haid, Ngoc Thang Vu, and Jonas Kuhn. 2019a. IMSurReal: IMS at the Surface Realization Shared Task 2019. In *Proceedings of the 2nd Workshop on Multilingual Surface Realisation (MSR 2019)*, pages 50–58, Hong Kong, China. Association for Computational Linguistics.

Xiang Yu, Agnieszka Falenska, Ngoc Thang Vu, and Jonas Kuhn. 2019b. Head-First Linearization with Tree-Structured Representation. In *Proceedings of the 12th International Conference on Natural Language Generation*, pages 279–289, Tokyo, Japan. Association for Computational Linguistics.

Mikhail Zaslavskiy, Marc Dymetman, and Nicola Cancedda. 2009. Phrase-Based Statistical Machine Translation as a Traveling Salesman Problem. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 333–341, Suntec, Singapore. Association for Computational Linguistics.

A Model Details and Hyperparameters

Our system is a modification on Yu et al. (2019a), where the encoder architecture and hyperparameters are identical to theirs, and we only change the decoder. The system is implemented with the DyNet library (Neubig et al., 2017). Training and testing are conducted on a single CPU core, where the average training time is under 1 hour and the average decoding speed is 50ms per sentence by the proposed model (Tree-LSTM encoder + subtree TSP decoder + swap post-processing).

Hyperparameter	Value
lemma dim	64
UPOS dim	32
morphological feature dim	32
dependency label dim	32
all other hidden dims	128
all LSTM layers	1
beam size	32
avg. token feature params	1.0×10^6
Tree-LSTM params	7.4×10^5
beam decoder params	1.6×10^6
TSP decoder params	6.6×10^4
swap decoder params	6.4×10^5
batch size	1
dropout	none
max training step	1×10^6
optimizer	Adam
Adam α	0.001
Adam β_1	0.9
Adam β_2	0.999

Table 7: Model hyperparameters.