Recurrent Models and Lower Bounds for Projective Syntactic Decoding

Natalie Schluter
IT University of Copenhagen
Copenhagen, Denmark
natschluter@itu.dk

Abstract

The current state-of-the-art in neural graph-based parsing uses only approximate decoding at the training phase. In this paper aim to understand this result better. We show how recurrent models can carry out projective maximum spanning tree decoding. This result holds for both current state-of-the-art models for shift-reduce and graph-based parsers, projective or not. We also provide the first proof on the lower bounds of projective maximum spanning tree, DAG, and digraph decoding.

1 Introduction

For several years, the NLP field has seen widespread investigation into the application of Neural Networks to NLP tasks, and with this, much, rather inexplicable progress. A string of very recent work (for example, Chen et al. (2018); Weiss et al. (2018); Peng et al. (2018)), has attempted to delve into the formal properties of neural network topology choices, in attempts to both motivate, predict, and explain associated research in the field. This paper aims to further contribute along this line of research.

We present the results of our study into the ability of state-of-the-art first-order neural graph-based parsers, with seemingly simple architectures, to explicitly forego structured learning and prediction.[1] In particular, this is not due to a significantly faster, simpler, algorithm for projective maximum spanning tree (MST) decoding than Eisner (1996)'s algorithm, which we formally prove to be impossible, given the Exponential Time Hypothesis. But rather, this is due to the capacity of recurrent components of these architectures to implicitly discover a projective MST. We prove this formally by showing how these recurrent components can intrinsically simulate exact projective decoding.

The context. The current state-of-the-art for graph-based syntactic dependency parsing is a seemingly basic neural model by Dozat and Manning (2017). The parser's performance is an improvement on the first, even simpler, rather engineering-free, neural graph-based parser by Kiperwasser and Goldberg (2016). This latter parser updates with respect to an output structure: projective decoding over a matrix of arc scores coupled with hinge loss between predicted and gold arcs, reporting parser performance of, for example, 93.32% UAS and 91.2% LAS on the converted Penn Treebank.[2] Remarkably, the former parser by Dozat and Manning (2017) forgoes entirely any structural learning, employing simple cross-entropy at training time, and saving (unconstrained) maximum spanning tree decoding for test time.

We further optimised Kiperwasser and Goldberg (2016)'s parser (Varab and Schluter, 2018) and extended it for cross-entropy learning, as is done by Dozat and Manning (2017). At test time, instead of any explicit decoding algorithm over the arc score matrix, we simply take the maximum weighted incoming arc for each word; that is, the parser is highly streamlined, without any heavy neural network engineering, but now also without any structured learning, nor without any structural decoding at test time. The resulting neural parser still achieves an impressively competitive UAS of 92.61% evaluated on the converted Penn Treebank data, without recourse to any pre-trained embeddings, unlike the systems by Kiperwasser and Goldberg (2016) and Dozat and

[1] For the remainder of this paper, all decoding algorithms discussed are first-order.

[2] Training is on Sections 2-21, development on Section 22 and testing on Section 23), converted to dependency format following the default configuration of the Stanford Dependency Converter (version $\geq$ 3.5.2).

Proceedings of NAACL-HLT 2019, pages 251–260
Minneapolis, Minnesota, June 2 - June 7, 2019.

Manning (2017). Using GloVe 100-dimensional Wikipedia and Gigaword corpus (6 billion tokens) pretrained embeddings, without updates, but linearly projected through a single linear dense layer to the same dimension, the structure-less parser achieves 93.18% UAS.[3] With this paper, we shed light on these surprising results from seemingly simple architectures. The insights we present here apply to any neural architecture that first encodes input words of a sentence using some type of recurrent neural network–i.e., all current state-of-the-art graph-based or shift reduce neural parsers.

Our contributions. This paper presents results for understanding the surprisingly superior performance of structure-free learning and prediction in syntactic (tree) dependency parsing.

1. We provide a formal proof that there will never be an algorithm that carries out projective MST decoding in sub-cubic time, unless a widely believed assumption in computational complexity theory, the *Exponential Time Hypothesis* (ETH), is false.

   Hence, computationally, we provide convincing evidence that these neural parsing architectures cannot be as simple as they appear. These results are then extended to projective maximum spanning DAG and digraph decoding.
2. In particular, we then show how to simulate Eisner's algorithm using a single recurrent neural network. This shows how, in particular, the LSTM stacked architectures for graph-based parsing by Dozat and Manning (2017), Cheng et al. (2016), Hashimoto et al. (2017), Zhang et al. (2017), and Kiperwasser and Goldberg (2016), are capable of intrinsically decoding over arc scores.

   This therefore provides one practical application where RNNs do not need supplementary approximation considerations (Chen et al., 2018).

2 Preliminaries

The Exponential Time Hypothesis (ETH) and k-Clique. The Exponential Time Hypothesis is a widely held though unproven computational hardness assumption stating that 3-SAT (or any of the several related NP-complete problems) cannot be solved in sub-exponential time in the worst case (Impagliazzo and Paturi, 1999). According to ETH, if 3-SAT were solvable in sub-exponential time, then also $P = NP$. But the ETH assumption is stronger than the assumption that $P \neq NP$, so the converse is not necessarily true. ETH can be used to show that many computational problems are equivalent in complexity, in the sense that if one of them has a subexponential time algorithm then they all do.

The k-Clique problem is the parameterised version of the NP-hard Max-Clique problem. This canonical intractable problem in parameterised complexity asks, given an input graph, whether there exists a clique of size k. A naïve algorithm for this problem running in $O(n^k)$ time checks all n^k combinations of nodes and verifies each combination in $O(k^2)$ time to see if they form a clique. However, Chen et al. (2006) showed that the problem has no $n^{o(k)}$ time algorithm–that is, the problem has no algorithm that runs in time subexponential in the exponent k assuming ETH.[4].

Recurrent neural networks. Recurrent neural networks (Rumelhart et al., 1986), as we generally use them in practise in NLP, take as input a matrix $\mathbf{x}$, containing a sequence of n vectors $\mathbf{x} = \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}$, and apply the following set of equations recursively, with h_0 the initial state:

$$\mathbf{h_t} = g(\mathbf{b} + \mathbf{W}\mathbf{h_{(t-1)}} + \mathbf{U}\mathbf{x_t})$$

Here, g is the activation function. Typically this activation function is $\tanh$, however the computational power of the model is theoretically maintained with any so-called "squashing" function (Siegelmann, 1996).

The choice of g, on the other hand, has been shown to affect the power of the recurrent model in general, depending on the restrictions involved in the formal investigation. For the purposes of this paper, the activation function is a rectified linear unit, or `ReLU`. The general computational power of such RNNs has recently been formally explored by Chen et al. (2018) (given infinite precision) and Weiss et al. (2018) (given finite precision), and empirically investigated for practical considerations of convergence under training by Le et al. (2015).

[3] Our structure-less but optimised implementation of the (Kiperwasser and Goldberg, 2016) graph-based parser, with 100 dimensional generated word embeddings, 50 dimensional generated POS-tag embeddings, a stack of 3 BiLSTMs with an output dimension of 225 each (total 450 concatenated), no dropout, MLP mappings for arc nodes of 400 dimension, and for labels of 100 dimensions. We use DyNet 2.1 (Neubig et al., 2017), and the parser code is freely available at `https://github.com/natschluter/MaxDecodeParser`.

[4] The problem is said to be $W[1]$-complete (Flum and Grohe, 2006)

LSTMs (Hochreiter and Schmidhuber, 1997) are RNNs with weighted self-loops (so-called gates). The recurrence equations take the form:

$$\mathbf{f_t} = g_1(\mathbf{b_f} + \mathbf{W_f h_{(t-1)}} + \mathbf{U_f x_t})$$
$$\mathbf{i_t} = g_1(\mathbf{b_i} + \mathbf{W_i h_{(t-1)}} + \mathbf{U_i x_t})$$
$$\mathbf{o_t} = g_1(\mathbf{b_o} + \mathbf{W_o h_{(t-1)}} + \mathbf{U_o x_t})$$
$$\mathbf{c_t} = f_t \circ c_{t-1} + i_t \circ g_1(\mathbf{b_c} + \mathbf{W_c h_{(t-1)}} + \mathbf{U_c x_t})$$
$$\mathbf{h_t} = \mathbf{o_t} \circ g_2(c_t)$$

where g_1, g_2 are activation functions. Setting all $\mathbf{W_f}, \mathbf{W_i}, \mathbf{W_c}, \mathbf{U_f}, \mathbf{U_i}, \mathbf{U_c}$ to be zero matrices, b_f to be a 0 vector, b_i, b_c to be 1 vectors, and the activation function g_2 to be `ReLU` we see that, in terms of hidden states, the LSTM model includes that of the RNN. In this paper, all activation functions are `ReLUs`.

3 Related Work

State-of-the-art in neural syntactic dependency parsing. The graph-based neural architectures we refer to here have important commonalities. We focus our discussion on the key contributions by Kiperwasser and Goldberg (2016) (the simplest architecture, and the first), and by Dozat and Manning (2017) (the state-of-the-art).

The architectures can be partitioned into three general components:

1. **Word representation generation**: Both architectures generate word embeddings and POS-tag embeddings. Pretrained embeddings, if they are being used, are added to the trained embeddings, and concatenated to corresponding POS-tag embedding. The embeddings are sent through a stacked BiLSTM. Output embeddings are projected to two further vector representations: as head node or as dependent node (specialised representations).
2. **Arc scoring**: All (head, dependent) combinations are scored.
3. **Decoding**: By some decoding process, the arc score matrix yields a (possibly disconnected) graph representation of the input sentence: (n-1) arcs, where no word has more than one head, as well as their probabilities.

We show in this paper how the second and third components can be carried out implicitly within the BiLSTM layers of the first component. Since currently state-of-the-art shift-reduce parsers also encode input words of a sentence using some type of recurrent neural network, this insight also applies to these non-graph-based models.

Related computational hardness results. To date there is no known truly sub-cubic algorithm for Boolean Matrix Multiplication (BMM), nor for Context-Free Grammar (CFG) parsing. Adapting Satta (1994)'s lower bound proof for Tree Adjoining Grammar parsing, Lee (1997) proved that BMM can be reduced to finding a valid derivation a string of length $O(n^{\frac{1}{3}})$ with respect to a CFG of size $\Theta(n^2)$. Lee (1997)'s reduction shows that there can be a no $O(|G|n^{3-\epsilon})$ for some constant $\epsilon > 0$ (sub-cubic-time) algorithm for CFG-parsing without implying a significant breakthrough in BMM, which is widely believed not to be possible. However, the construction required the grammar size $|G| = \Theta(n^6)$ to be dependent on the the input size n, which, as Lee (1997) points out, is unrealistic in most applications.

Abboud et al. (2015), on the other hand, present a proof of the unlikelihood of a sub-cubic algorithm for CFG-parsing using ETH and specifically the k-Clique problem. Given an instance of the $3k$-Clique problem (i.e., an undirected graph and the parameter $3k$), they construct a string w of length n^k and a CFG, G of constant size (for any $3k$) such that if G derives w in sub-cubic time, then there is an algorithm running in time $n^{o(3k)}$ for the $3k$-Clique problem, which, as we explained in Section 2, is impossible, assuming ETH.

To date, no truly sub-cubic algorithm for projective maximum spanning tree decoding is known. In the next section, we present a proof similar in spirit to Abboud et al. (2015)'s that also shows that such an algorithm most likely cannot be found.

4 Lower Bounds for First-Order Projective Dependency Decoding

Current state-of-the-art neural graph-based parsers forego structural learning and do not even seem to require structured prediction. In this section, we provide evidence that this is indeed not because the parsers are so seemingly simple. Computationally it is unlikely that some simpler and faster decoding method alone is achieving such a competitive performance. We show this with the following theorem.

Theorem 1. *Under the assumption of ETH, there is no algorithm that carries out projective MST decoding in time significantly faster than $O(n^3)$; that is, there is no sub-cubic ($O(n^{3-\epsilon})$ for some constant $\epsilon > 0$) time algorithm for finding the maximally weighted projective spanning tree, T^*, over a weighted digraph input.*

Notation and special remarks. We denote by $[n]$ the set $\{1, \ldots, n\}$. For lack of a better symbol, we use $\odot$ here to signify iterative string concatenation, which otherwise is signified by just writing symbols beside each other, or by the symbol $\cdot$. Rather than working over words of a sentence, given the formal nature of the proof, the projective MST algorithm must work over symbols of the input word w. Hence the input is a weighted digraph over the symbols of w and the output is a projective MST, T^*, over these symbols. The reduction, makes use of the weight of T^*.

Proof (of Theorem 1). Let $G = (V, E)$ be an arbitrary simple undirected graph. We place an arbitrary order on the nodes from V and fix it, so $V := \{v_1, \ldots, v_n\}$.

As in Abboud et al. (2015)'s reduction from $3k$-clique to CFG-parsing, we first generate a string of length $O(n^k)$ to represent the graph for the task at hand; we do so in $O(n^k)$ time. The string contains a representation of all of the possible k-cliques in the graph. We can create a listing of all of these k-cliques using exhaustive search in at most $O(n^k)$ time and space. Let $K := \{\{v_{i_1}, \ldots, v_{i_k}\}$ a k-clique in $G \mid i_j \in [n], v_{i_j} \in V\}$ correspond to the set of k-cliques from G, and place an arbitrary order on $K := \{k_1, \ldots, k_{|K|}\}$. So, $|K| \in O(n^k)$. We define $6 \cdot k \cdot |K|$ sets of symbols with respect to V, each with $n(= |V|)$ elements:

- **Unmarked symbols**: $A_{i,t} := \{a_{i,j,t} \mid j \in [n]\}$ for $i \in [k], t \in [|K|]$ where $a_{i,j,t}$ corresponds to node $v_j \in V$.
  Similarly for the sets $B_{i,t}$ and $C_{i,t}$.
- **Marked symbols**: $\overline{A}_{i,t} := \{\overline{a}_{i,j,t} \mid a_{i,j,t} \in A_i\}$. Similarly for the sets $\overline{B}_{i,t}$ and $\overline{C}_{i,t}$.

We let $A = \cup_{i\in[k],t\in[|K|]}(A_{i,t} \cup \overline{A}_{i,t})$ and similarly for B and C. Then the vocabulary for constructing our input word is $U := A \cup B \cup C$.

Constructing the input word w. We now construct a word w over the vocabulary U such that if the projective maximum spanning tree has weight $|w| + 2k - 2 + |K|$, then the graph G has a $3k$-clique. We do this by defining the weights of possible arcs between carefully selected pairs of symbols from the vocabulary. The entire construction of the word w takes time $O(n^k)$ (coinciding with the upper bound on the word's length).

The input word is made up of a series of gadgets. For each k-clique, we have three types of gadgets: A-, B-, and C-gadgets. A- and C-gadgets each correspond both to a particular k-clique in G, as well as all k-cliques in G. B-gadgets, on the other hand, only correspond to particular k-cliques in G. Let $k_t = \{v_{(t,1)}, \ldots, v_{(t,k)}\} \in K$ be the tth k-clique. Even if each $v_{(t,q)}$ is a node in $V(G)$ the notation for indices is useful to refer to the qth node of the tth k-clique. Also, in what follows, we use the middle index of symbols to simultaneously refer to the k-clique membership: $j_{(t,q)} \in [n]$, and simultaneously allows us to refer to the qth node in the tth k-clique from K, for $q \in [k], t \in [|K|]$.

A-gadgets:

$$A(t) := \odot_{i\in[k]}(\overline{a}_{i,j_{(t,1)},t}\overline{a}_{i,j_{(t,2)},t} \cdots \overline{a}_{i,j_{(t,k)},t}) \cdot \odot_{i\in[k]}(a_{i,j_{(t,1)},t} \cdot a_{i,j_{(t,2)},t} \cdots a_{i,j_{(t,k)},t})$$

C-gadgets:

$$C(t) := (\odot_{i\in[k]} c_{i,j_{(t,1)},t} c_{i,j_{(t,2)},t} \cdots c_{i,j_{(t,k)},t}) \cdot \overline{c}_{k,j_{(t,k)},t} \cdot \overline{c}_{k-1,j_{(t,k-1)},t} \cdots \overline{c}_{1,j_{(t,1)},t}$$

and **B-gadgets:**

$$B(t) := L_t b_{k,j_{(t,k)},t} \cdots b_{2,j_{(t,2)},t} b_{1,j_{(t,1)},t} H_t \overline{b}_{k,j_{(t,k)},t} \cdot \overline{b}_{k-1,j_{(t,k-1)},t} \cdots \overline{b}_{2,j_{(t,2)},t} \overline{b}_{1,j_{(t,1)},t} R_t,$$

We call the symbol H_t the *head of the gadget* $B(t)$, and L_t and R_t the gadget's *left and right boundary symbols* respectively.

We then set the word w to be

$$\big(\odot_{t\in[|K|]} A(t)\big) \cdot \big(\odot_{t\in[|K|]} B(t)\big) \cdot \big(\odot_{t\in[|K|]} C(t)\big)$$

consisting of an *A-gadget region* followed by a *B-gadget region*, and then a *C-gadget region*.

Idea of the proof. The idea of the proof is to allow an optimal projective MST, T^*, to be built that matches up one distinct (with respect to the k-clique) gadget from each region, each representing different k-cliques whenever there is a $3k$-clique in G. We will deduce the existence of such a clique by the weight of T^*. Essentially, a projective spanning tree of weight $|w| - 1$ will always be present, but T^* having weight superior to this will indicate a matching up of gadgets. Now, suppose we have a sub-cubic projective MST algorithm A. By our construction, if A returns a T^* with weight $|w| + 2k - 2 + |K|$, then there is a $3k$-clique. Otherwise, there is no $3k$-clique. On input of length n^k, the sub-cubic time algorithm runs in time $O((n^k)^{3-\epsilon}) = O(n^{3k-k\epsilon}) \in n^{o(3k)}$ for some constant $\epsilon > 0$. Thus A will have solved $3k$-clique in time $n^{o(3k)}$, which is impossible under the ETH assumption.

Note that by the definition of a $3k$-clique, a $3k$-clique can be partitioned arbitrarily into 3 equal

sized sub-graphs over k nodes that must each form a k-clique. So, if $|K| < k$, then there trivially cannot be any $3k$-clique in G. We therefore only consider without loss of generality the argumentation for the case where $|K| \geq k$, since our algorithm can simply return a negative answer about the existence of a $3k$-clique in G after enumerating the set K and before computing any projective MST.

The projective MST algorithm takes as input the description of a weighted digraph, D, whose nodes are defined by symbols of the input word w. The digraph need not be explicitly constructed, since the algorithm can simply use the description of the digraph that follows instead to check for the existence of arcs between symbols. This description has constant length.

A description of the input weighted graph D over w. For the input digraph, arcs can (1) be missing from the fully complete digraph, (2) have weight 1, or (3) have weight 2. To construct D, weights are assigned to arcs by the following rules.

Weight 1 arcs. The following arcs of our input graph have weight 1.

1. **Region connectivity arcs.** These arcs ensure connectivity is possible within respective gadget regions.
   (a) All arcs $(\overline{a}_{1,j',t}, a_{i,j,t-1})$ and $(\overline{a}_{1,j',t}, \overline{a}_{i,j,t-1})$, i.e., the first symbol of the tth A-gadget attaches to all symbols of the previous ($t-1$th) A-gadget.
   (b) All arcs $(\overline{c}_{1,j',t}, c_{i,j,t+1})$ and $(\overline{c}_{1,j',t}, \overline{c}_{i,j,t+1})$, i.e., the last symbol of the tth C-gadget attaches to all symbols of the next ($t+1$th) C-gadget gadget.
   (c) All arcs $(b_{i,j,t}, b_{i+1,j',t})$ and $(\overline{b}_{i+1,j,t}, \overline{b}_{i,j',t})$ for $i \in [k-1]$.
   (d) All arcs $(b_{k,j,t}, L_t)$, $(\overline{b}_{1,j',t}, R_t)$.
   (e) All arcs $(H_t, b_{1,j,t})$ and $(H_t, \overline{b}_{k,j',t})$ making H_t a possible head of the respective B-gadget (*B-gadget heads*) for any MST.
   (f) All arcs (L_{t+1}, H_t), (R_t, H_{t+1}) for all $t \in [|K|-1]$.
   (g) All arcs $(c_{k,j_{(t,k)},t}, \overline{c}_{k,j,t})$, i.e., arcs from the last nonmarked symbol to the first marked symbol, in every C-gadget. Also, all arcs $(\overline{c}_{i+1,j,t}, \overline{c}_{i,j,t})$ for $i \in [k-1]$, i.e., together forming a path of marked symbols within each C-gadget.
   The following arcs are the reversals of (1c) through (1e).
   (h) All arcs $(b_{i+1,j,t}, b_{i,j',t})$ and $(\overline{b}_{i,j,t}, \overline{b}_{i+1,j',t})$ for $i \in [k-1]$.
   (i) All arcs $(L_t, b_{k,j,t})$, $(R_t, \overline{b}_{1,j',t})$.
   (j) All arcs $(b_{1,j,t}, H_t)$ and $(\overline{b}_{k,j',t}, H_t)$ making H_t the head of the respective B-gadget (*B-gadget heads*) for any MST.
2. **Boundary connectivity arcs.** These arcs ensure that the boundaries of regions are connected.
   (a) The arcs $(L_1, a_{i,j,|K|})$ and $(L_1, \overline{a}_{i,j,|K|})$, i.e., all symbols from the last of the A-gadgets attach to the first symbol of the B-gadget region.
   (b) The arcs $(R_{|K|}, c_{i,j,1})$ and $(R_{|K|}, \overline{c}_{i,j,1})$, i.e., all symbols from the first of the C-gadgets attach to the last symbol of the B-gadget region.
3. **G-induced arcs.** These arcs reflect the connections of the original graph G, and ultimately the existence of a $3k$-clique.
   (a) All arcs $(b_{i,j,t}, a_{i,j',t'})$, for each $i \in [k-1], t \neq t'$, if $v_j v_{j'} \in E(G)$ (i.e., not for $i = k$, which has a weight of 2 rather).
   (b) All arcs $(\overline{b}_{i,j,t}, c_{i,j',t'})$, for each $i \in \{2, \ldots, k\}, t \neq t'$, if $v_j v_{j'} \in E(G)$ (i.e., not for $i = 1$, which has a weight of 2 rather).
   (c) All arcs $(\overline{c}_{i,j,t}, \overline{a}_{i,j',t'})$ for all $i \in [k], t \neq t'$, if $v_j v_{j'} \in E(G)$ (i.e., this time also for $i = 1$).

As we show in Lemma 1.1, with the region connectivity arcs (1a-1g) and boundary connectivity arcs (2), we ensure that the algorithm can always return a projective MST with weight at least $|w| - 1$. The G-induced arcs and region connectivity arcs (1h-1j, 4) on the other hand will be triggered to use by the algorithm's prioritisation of the following arcs.

Weight 2 arcs. We have the following arcs of weight 2.

4. **Region connectivity arcs.** (L_{t+1}, L_t) and (R_t, R_{t+1}) for $t \in [|K|-1]$.
5. **G-induced arcs.**
   (a) All arcs $(b_{k,j,t}, a_{k,j',t'})$, for each $t \neq t'$, if $v_j v_{j'} \in E(G)$
   (b) All arcs $(\overline{b}_{1,j,t}, c_{1,j',t'})$, for each $t \neq t'$, if $v_j v_{j'} \in E(G)$

There are no other arcs in the input digraph D.

Lemma 1.1. *There always exists a projective MST in D of weight $|w| - 1$.*

Proof. The A-region, together with the symbol L_1 from the B-region can form a tree rooted in L_1

using region connectivity arcs (1a) with boundary connectivity arcs (2a)–all weight 1 arcs. Similarly for the C-region with the symbol $R_{|K|}$ from the B-region (arcs (1b) and (2b)). Moreover, these regional sub-trees are trivially projective. If we construct a projective subtree out of the B-region, in which L_1 and $R_{|K|}$ are leaf nodes, then we have the result.

The combination of weight-1 arcs from (1c), (1d), and (1e) results in each B-gadget B(t) being a projective subtree headed by its head node H_t made up of a combination of two paths $H_t, b_{1,j_1,t}, \ldots, b_{k,j_k,t}, L_t$ and $H_t, b_{k,j_k,t}, \ldots, \overline{b_{1,j_1,t}}, R_t$. To make a projective subtree out of the entire B-region, we choose some arbitrary H_t node as the root and take further weight-1 arcs described in (1f): (L_p, H_{p-1}) if $p \leq i$ and $(R_p, H_p + 1)$ otherwise, for $i \in [2, |K| - 1]$. In all these possible B-regional projective subtrees, both L_1 and $R_{|K|}$ are leaf nodes, which gives the result. □

Lemma 1.2. *Let T^* be a projective MST over D. There are at most $2k+(|K|-1)$ arcs of weight 2 in T^*: k from the B- to the A-region, k from the B- to the C-region, and the rest internal to the B-region.*

The number of arcs of weight 2, internal to or originating from the B-region, will be maximised if arcs exiting the B-region all originate from the same B-gadget (instead of 2+ distinct ones).

Moreover, suppose distinct $t_1, t_2, t_3 \in [|K|]$. If T^* includes an arc of weight 2 from gadget $B(t_2)$ to gadget $A(t_1)$ and from gadget $B(t_2)$ to gadget $C(t_3)$, then T^* must also include arcs characterised by the following

1. *all non-marked nodes in $A(t_1)$ have non-marked heads in $B(t_2)$,*
2. *all non-marked nodes in $C(t_3)$ have marked heads in $B(t_2)$, and*
3. *all marked nodes in $A(t_1)$ have marked heads in $C(t_3)$.*

Proof. There are only arcs of weight 2 in D from the B-region to both the A- and the C-regions, and internally in the B-region. We show that there are at most k weight 2 arcs connecting the A- and B-regions and C- and B-regions. Then we show that the maximal number of weight 2 edges internal to the B-region is $(|K| - 1)$.

Suppose there are more than k arcs of weight 2 from the B-region to the A-region in T^*. Then there are at least two of these arcs entering different A-gadgets: $(b_{1,j',t'}, a_{1,j,t})$ and $(b_{1,i',p'}, a_{1,i,p})$, with $p < t$. Consider the barred symbols in the tth A-gadget. There are only two possible heads: (1) the symbol following the gadget (region connectivity arcs (1a) or boundary connectivity arcs (2a)), which by projectivity is excluded because these arcs would cross $(b_{1,j',t'}, a_{1,j,t})$, or (2) symbols from the C-region, which by projectivity is also excluded because they would cross the arc $(b_{1,i',p'}, a_{1,i,p})$.

The proof that there are at most k arcs of weight 2 from the B-region to the C-region in T^* is analogous.

For the maximal number of arcs of weight 2, internal to the B-region, we first consider the maximum number of weight 2 region connectivity arcs (4). By projectivity, a B-gadget with arcs entering an A- or C-gadget cannot have any entering weight 2 region connectivity arc. Also, by projectivity, a single B-gadget can have at most 1 weight 2 region connectivity arc. Thus, the number of weight 2 arcs would be maximised by ensuring arcs exiting the B-region originate from the same B-gadget, so only one B-gadget does not have weight 2 entering arcs. Since there are $|K|$ B-gadgets in total, this means there are at most $|K| - 1$ weight 2 B-region internal arcs.

The rest of the proof follows by the similar projectivity arguments. □

Lemma 1.3. *T^* has weight $|w|+2k-1+(|K|-1)$ if and only if there is a $3k$-clique in G.*

Proof. ($\Leftarrow$) Suppose there is a $3k$-clique in G consisting of the three k-cliques k_1, k_2, and k_3, and such that $k_1 \cup k_2 \cup k_3$ is a $3k$-clique. In w, there must be corresponding gadgets in each of its gadget regions. We consider $A(1)$ the gadget for k_1 in A, by $B(2)$ the gadget for k_2 in B, and by $C(3)$ the gadget for k_3 in C. We build up a set S of arcs based on these three gadgets. The set S consists of all the possible G-induced (weight 1 and 2) arcs between these three regions–a disconnected set where no two arcs cross, by Lemma 1.2. By the same lemma, S includes exactly $2k + (|K| - 1)$ arcs of weight 2. We will add arcs to S to connect the rest of the symbols in w until we form a tree, and by Lemma 1.2 again we cannot add any further weight 2 arcs.

We must now supplement S to make a tree. We first connect the B-region. For $t < 2$, we connect B-gadgets internally by making the path from the R_t to L_t, using weight 1 region connectivity arcs. We make paths in the opposite direction, from L_t

to R_t for $t > 2$. We then add all possible weight 2 region connectivity arcs. This makes A(1) and B-region connected.

All other A-gadgets are connected as in the proof of Lemma 1.1. Similarly for the C-gadgets before and after $C(3)$.

The only nodes that still lack a head node are the marked nodes from $C(3)$. We connect these using region connectivity arcs from (1g).

We have now constructed a projective tree of weight $|w| - 1 + 2k + (|K| - 1)$. We cannot have a higher weighted projective tree by Lemma 1.2. Hence tree is an optimal T^*.

($\Rightarrow$) Suppose T^* has weight $|w| - 1 + 2k + (|K| - 1)$. By Lemma 1.2, T^* has exactly k arcs of weight 2 from the B-region to the A-region, and k from the B-region to the C-region, and that in this case all possible G-induced arcs between the three corresponding gadgets are in T^*. Moreover, internally to the B-region, there are $|K|-1$ weight 2 edges.

Let the gadgets be w.l.o.g., A(1), B(2), and C(3). Each unmarked b symbol in B(2) corresponds to a node in V, and is the head of an unmarked symbol from A(1) corresponding to every node in k-clique k_1. This means that in G, all possible connections between nodes in k_1 and k_2 exist. The same holds for B(2) with C(3) and C(3) with A(1). Hence there is a $3k$-clique in G. □
□

With Theorem 1, we have shown that the non-structural graph-based neural parsing systems cannot be carrying out explicit exact decoding in with a significantly simpler algorithm. As we show in the next section, in fact, the LSTM stacks of these systems alone are powerful enough to simulate all components.

In our proof, the algorithm consistently makes a choice between edges of weight 1 and edges of weight 2 for the result to preserve projectivity. Possibly more edges of weight 1 may end up in a maximum spanning projective DAG or digraph, so we cannot necessarily use the weight in the same way to deduce the result. The number of edges in D is less than n^2. Hence if we replace the weights of weight 1 arcs in D by weight $1/(n^2)$, then an output maximum spanning projective digraph or DAG with weight superior to $2k+(|K|-1)$ would indicate a $3k$-clique. By the algorithms to do this from (Schluter, 2015) in cubic time, we therefore have the same lower bound for finding a maximum spanning projective DAG or digraph.

Corollary 1.1. *Under the assumption of ETH, there is no algorithm that carries out projective maximum spanning DAG or digraph decoding in sub-cubic time.*

5 RNN Simulation of Eisner's Algorithm

Eisner (1996)'s algorithm on an input sentence of length n uses an $n \times n$ table M and dynamic programming to compute for table cell $M_{i,j}$ the highest weighted sub-trees over the span (i, j) of the input sentence. The algorithm iterates over spans of increasing length. For $M_{i,j}$, the weights of all possible combinations of sub-spans are considered as candidate sub-trees over the span, and the maximum of these is retained in $M_{i,j}$.

For our purposes, the problem with this version of the algorithm is that the RNN cannot compute the maximum of the corresponding $O(n)$ values in either constant space nor in one time-step, and the corresponding sub-tree weight is required in the computation of maximum sub-trees over the span $j - i + 1$ at the next recursive step.

In Algorithm 1, we precompute enough of the comparisons required for finding the maximum spanning sub-tree combination before the algorithm arrives in that table cell (from line 5). Thus, instead of taking the maximum across $k \in O(n)$ values, we only ever take the maximum across 2 values at a time. We now explain this algorithm.

A sub-tree over the span (i, j) is said to be *complete* if it includes some arc between i and j. Otherwise the sub-tree is called *incomplete*. We use seven weight matrices (which we extend to 25 matrices later):

- S an $n \times n$ matrix of arc scores, where $S[i, j]$ is the score of the (i, j) arc.
- I an $n \times n$ matrix of incomplete sub-tree scores, where $I[i, j, h]$ is the incomplete sub-tree for the span (i, j) with head $h \in 0, 1$. If $h = 0$, then i is the root of the sub-tree, and if $h = 1$, then j is the root.
- C is defined in the same way as I but for complete sub-trees.
- $I_r[i, j, h]$ (resp. C_r) stores the current "row"-maximum value for $I[i, j, h]$ across the span combinations $(i, k), (k + 1, j)$ for $k - i > (k + 1) - j$ (resp. $(i, k), (k, j)$ for $k - i > k - j$). These are the cases where the span (i, k) is the largest of the two sub-spans (i, j). These table values are adjusted while the algorithm visits cells (i, k).

- $I_c[i,j,h]$ (resp. C_c) stores the current "column"-maximum value for $I[i,j,h]$ across the span combinations $(i,k),(k+1,j)$ for $k-i \leq k+1-j$ (resp. $(i,k),(k,j)$ for $k-i > k-j$). These are the cases where the span $(k+1,j)$ (resp. (k,j)) is larger or equal to the other sub-span of the partitioned span (i,j). These table values are adjusted while the algorithm visits cells $(k+1,j)$ (resp. (k,j)).

The pseudocode for this algorithm, which we refer to as `streaming-max-eisner` is presented in Algorithm 1. The main difference with the original version is that the internal loop partitioning of a span is separated in Algorithm 1 over several previous iterations of the loop, so that once the algorithm visits cell (i,j), all that needs to be computed is the maximum of the two row- and column-maximum values, from I_r and I_c, or from C_r and C_c.

It is straightforward to show the correctness of this algorithm, which we state as Theorem 2. We omit the proof due to space constraints. The algorithm can also be easily adapted for backtracking.

Theorem 2. *Algorithm 1 returns the weight of T^*.*

We make a final adjustment to the algorithm before stating the simulation construction. For the simulation, we only have RNN operations at our disposal: linear combinations and a $\texttt{ReLU}$ activation function, but no explicit $\max$ operation. In order to use only RNN operations, we replace the explicit max function.

Replacing the explicit max function. We note that to find the maximum of the two positive numbers a and b, we can use the $\texttt{ReLU}$ function. Without loss of generality, suppose that $a > b$, then

$$\begin{aligned}\frac{(\texttt{ReLU}(a-b)+\texttt{ReLU}(b-a)+a+b)}{2} \\ = \frac{(a-b+a+b)}{2} = \frac{2a}{2} = a \\ = \max(a,b). \quad (1)\end{aligned}$$

In fact, since all weights are assumed positive, Equation 1 can be rewritten as

$$\begin{aligned}\max(a,b) = \frac{1}{2}(\texttt{ReLU}(a-b)+\texttt{ReLU}(b-a) \\ +\texttt{ReLU}(a)+\texttt{ReLU}(b)). \quad (2)\end{aligned}$$

We therefore make a final adjustment to the original Eisner algorithm, over the version Algorithm 1, replacing all $\max$ functions using Equation 2. Instead of storing only one value for each matrix I_r, I_c, C_r, C_c, I, C, we store four, denoted by the fields a, b, ab, ba corresponding the four values we need to store: $\texttt{ReLU}(a), \texttt{ReLU}(b), \texttt{ReLU}(a-b)$, and $\texttt{ReLU}(b-a)$ respectively. For instance, for the matrix I, we have I_a, I_b, I_{ab}, I_{ba}. Then, for example, line 6 becomes

$$\begin{aligned}a \leftarrow \texttt{ReLU}(\frac{1}{2} * (I_r[i,j,0].a + I_r[i,j,0].b \\ + I_r[i,j,0].ab + I_r[i,j,0].ba)) \quad (3)\\ b \leftarrow \texttt{ReLU}(\frac{1}{2} * (I_c[i,j,0].a + I_c[i,j,0].b \\ + I_c[i,j,0].ab + I_c[i,j,0].ba)) \quad (4)\end{aligned}$$

$$\begin{aligned}I[i,j,0].a &\leftarrow \texttt{ReLU}(a)\\ I[i,j,0].b &\leftarrow \texttt{ReLU}(b)\\ I[i,j,0].ab &\leftarrow \texttt{ReLU}(a-b)\\ I[i,j,0].ba &\leftarrow \texttt{ReLU}(b-a)\end{aligned}$$

where Equations 3 and 4 are wrapped in an extra $\texttt{ReLU}$ operation which yields no difference to the parameter, but which will be convenient for our simulation in Section 5.

Lines 11-14 and 16-19 are adapted in the same way. We provide the adaption of line 11 to make this precise:

$$\begin{aligned}a \leftarrow &\texttt{ReLU}(\frac{1}{2} * (I[i,p,1].a + I[i,p,1].b \\ &+ I[i,p,1].ab + I[i,p,1].ba)) \\ b \leftarrow &\texttt{ReLU}(\frac{1}{2} * (C[i,j,0].a + C[i,j,0].b \\ &+ C[i,j,0].ab + C[i,j,0].ba \\ &+ C[j+1,p,1].a + C[j+1,p,1].b \\ &+ C[j+1,p,1].ab + C[j+1,p,1].ba \\ &+ S[p,i]))\end{aligned}$$

$$\begin{aligned}I_r[i,p,1].a &\leftarrow \texttt{ReLU}(a)\\ I_r[i,p,1].b &\leftarrow \texttt{ReLU}(b)\\ I_r[i,p,1].ab &\leftarrow \texttt{ReLU}(a-b)\\ I_r[i,p,1].ba &\leftarrow \texttt{ReLU}(b-a).\end{aligned}$$

Algorithm 1 therefore uses 25 matrices on input of length n–hence, still $O(n^2)$ space.

Simulating Algorithm 1. The projective dependency parsing architecture M' to be simulated first sends word embeddings $x_i, i \in [n]$ through a forward (and backward) LSTM with output word representations $\overrightarrow{o}'_i$ (and $\overleftarrow{o}'_i$) of dimension d. The concatenated result $[\overrightarrow{o}'_i; \overleftarrow{o}'_i]$ is further specialised through two unrelated nonlinear dense layers: one for dependents and one for heads. Then all resulting pairs (dependent,head) of word representations are sent through a scoring function to generate a score matrix as input to projective MST decoding (Kiperwasser and Goldberg, 2016).

Algorithm 1 Projective MST algorithm computing the maximum over at most 2 arguments.

1: **procedure** STREAMING-MAX-EISNER
2: $S, I, C, I_r, I_c, C_r, C_c$ all initialised to 0 matrices
3: **for** $t \leftarrow 1$ to n-1 **do** ▷ span for-loop
4: **for** $i \leftarrow 1$ to $n - t$ **do** ▷ diagonal for-loop
5: $j \leftarrow i + t$ ▷ the algorithm is visiting cells (i, j)
6: $I[i, j, 0] \leftarrow \max(I_r[i, j, 0], I_c[i, j, 0])$
7: $I[i, j, 1] \leftarrow \max(I_r[i, j, 0], I_c[i, j, 1])$
8: $C[i, j, 0] \leftarrow \max(C_r[i, j, 0], C_c[i, j, 0])$
9: $C[i, j, 1] \leftarrow \max(C_r[i, j, 0], C_c[i, j, 1])$
10: **for** $p \leftarrow j + 1$ up to $\min(j + t + 1, n)$ **do** ▷ streaming-row for-loop, i.e., while $p - (j + 1) \leq t$
11: $I_r[i, p, 1] = \max(I[i, p, 1], C[i, j, 0] + C[j + 1, p, 1] + S[p, i])$
12: $I_r[i, p, 0] = \max(I[i, p, 0], C[i, j, 0] + C[j + 1, p, 1] + S[i, p])$
13: $C_r[i, p, 1] = \max(C[i, p, 1], C[i, j, 1] + I[j + 1, p, 1])$
14: $C_r[i, p, 0] = \max(C[i, p, 0], I[i, j, 0] + C[j + 1, p, 0])$
15: **for** $p \leftarrow i - 1$ down to $\max(i - 1 - t, 1)$ **do** ▷ streaming-column for-loop, i.e., while $(i - 1) - p \leq t$
16: $I_c[p, j, 1] = \max(I[p, j, 1], C[p, i - 1, 0] + C[i - 1, j, 1] + S[p, i])$
17: $I_c[i, p, 0] = \max(I[i, p, 0], C[i, j, 0] + C[j + 1, p, 1] + S[i, p])$
18: $C_c[i, p, 1] = \max(C[i, p, 1], C[i, j, 1] + I[j + 1, p, 1])$
19: $C_c[i, p, 0] = \max(C[i, p, 0], I[i, j, 0] + C[j + 1, p, 0])$
20: Return I, C

The architecture M to simulate M' consists of two components, each being a recurrent layer: a BiLSTM (for contextual word representations and word specialisations) and an RNN (for scoring and to simulate Algorithm 1).

M starts by feeding word embeddings x_i into its first component, the BiLSTM. In the forward direction, at the tth time step, the contextual representation $\overrightarrow{o}'_t$ is generated, $\overrightarrow{o}'_{t-1}$ is specialised to $\overrightarrow{o}^h_{t-1}$ (head) and $\overrightarrow{o}^d_{t-1}$ (dependent), and the previously specialised word representations in $\overrightarrow{o}_t$ (i.e., corresponding to $\overrightarrow{o}'_1, \ldots, \overrightarrow{o}'_{t-2}$) are copied over. We add a single extra $(n + 1)$th time step to each direction, so M can finish specialising contextualised word representations within this first component. Similarly for the backward direction.

There is one single input to M's second component, an RNN, which also works in $n + 1$ time steps. We refer to the inputs for this component as $z_1, \ldots, z_{(n+1)}$, where $z_2 \ldots z_{(n+1)}$ are all dummy inputs. z_1 is the concatenation of the final output vectors from each direction of M's BiLSTM. In the first time step of this component, M computes the score matrix and stores it in the hidden state h_1. The hidden state has a dimension large enough to house the 25 tables ($O(n^2)$ space) required by Algorithm 1 for subtree score bookkeeping and computing the maximum of two values using linear combinations and a `ReLU`.

The outer loop (the *span for-loop* with variable t) of the algorithm corresponds to each time-step t of the RNN. For the first internal for-loop (the *diagonal for-loop* with variable i), we note that, in lines 6-9, no cells $(i, i + t)$ whose values are being computed require information from each other at this time-step t.

The *streaming-row* and *streaming-column* for loops (lines 11-14, 16-19) on the other hand sometimes requires maximal values $(i, i + t)$ from lines 6-9 to be computed. This problem is simply solved by replacing the corresponding expressions appearing as left-hand sides in lines 6-9 by the right-hand sides.

The output h_{n+1} contains the desired maximum value.

6 Concluding Remarks

Recent state-of-the art neural graph-based parsers comprising, among other components, a short stack of BiLSTMs, seem to obviate any explicit structural learning or prediction. In this paper, under the assumption of ETH, we showed that this is not due to any possible indirect discovery of a faster algorithm for finding a projective maximum spanning tree and extended the result to projective maximimum spanning DAGs and digraphs. We further showed how these architectures allow for simulating decoding, implying that they are indeed carrying out implicit structured learning and prediction.

Acknowledgments

We gratefully acknowledge the careful remarks of the anonymous NAACL reviewers.

References

Amir Abboud, Arturs Backurs, and Vanessa Vassilevska Williams. 2015. If the current clique algorithms are optimal, so is Valiant's parser. In *Proceedings of FOCS*.

Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. 2006. Strong computational lower bounds via parameterized complexity. *Journal of Computer and System Sciences*, 8:1346–1367.

Yining Chen, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight. 2018. Recurrent neural networks as weighted language recognizers. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2261–2271, New Orleans, Louisiana. Association for Computational Linguistics.

Hao Cheng, Hao Fang, Xiaodong He, Jianfeng Gao, and Li Deng. 2016. Bi-directional attention with agreement for dependency parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2204–2214, Austin, Texas. Association for Computational Linguistics.

Timothy Dozat and Christopher M. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *Proceedings of ICLR*.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, COLING '96, pages 340–345, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jörg Flum and Martin Grohe. 2006. *Parameterized Complexity Theory*. Springer-Verlag.

Kazuma Hashimoto, caiming xiong, Yoshimasa Tsuruoka, and Richard Socher. 2017. A joint many-task model: Growing a neural network for multiple nlp tasks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1923–1933, Copenhagen, Denmark. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Russell Impagliazzo and Ramamohan Paturi. 1999. The complexity of k-SAT. In *Proceedings of the 14th IEEE Conference on Computational Complexity*, pages 237–240.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the ACL*, 4:313–327.

Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. 2015. A simple way to initialize recurrent networks of rectified linear units. *CoRR*, abs/1504.00941.

Lillian Lee. 1997. Fast context-free parsing requires fast boolean matrix multiplication. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 9–15, Madrid, Spain. 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*.

Hao Peng, Roy Schwartz, Sam Thomson, and Noah A. Smith. 2018. Rational recurrences. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1203–1214, Brussels, Belgium. Association for Computational Linguistics.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature*, 323:533–536.

Giorgio Satta. 1994. Tree-adjoining grammar parsing and Boolean matrix multiplication. *Computational Linguistics*, 20:173–191.

Natalie Schluter. 2015. The complexity of finding the maximum spanning DAG and other restrictions for DAG parsing of natural language. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics*, pages 259–268, Denver, Colorado. Association for Computational Linguistics.

Hava T. Siegelmann. 1996. Recurrent neural networks and finite automata. *Computational Intelligence*, 12:567574.

Daniel Varab and Natalie Schluter. 2018. Uniparse: A universal graph-based parsing toolkit. *CoRR*, abs/1807.04053.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the practical computational power of finite precision rnns for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia. Association for Computational Linguistics.

Xingxing Zhang, Jianpeng Cheng, and Mirella Lapata. 2017. Dependency parsing as head selection. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 665–676, Valencia, Spain. Association for Computational Linguistics.