Parsing with Traces: An $O(n^4)$ Algorithm and a Structural Representation

Jonathan K. Kummerfeld and **Dan Klein**
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720, USA
{jkk,klein}@cs.berkeley.edu

Abstract

General treebank analyses are graph structured, but parsers are typically restricted to tree structures for efficiency and modeling reasons. We propose a new representation and algorithm for a class of graph structures that is flexible enough to cover almost all treebank structures, while still admitting efficient learning and inference. In particular, we consider directed, acyclic, one-endpoint-crossing graph structures, which cover most long-distance dislocation, shared argumentation, and similar tree-violating linguistic phenomena. We describe how to convert phrase structure parses, including traces, to our new representation in a reversible manner. Our dynamic program uniquely decomposes structures, is sound and complete, and covers 97.3% of the Penn English Treebank. We also implement a proof-of-concept parser that recovers a range of null elements and trace types.

1 Introduction

Many syntactic representations use graphs and/or discontinuous structures, such as traces in Government and Binding theory and f-structure in Lexical Functional Grammar (Chomsky 1981; Kaplan and Bresnan 1982). Sentences in the Penn Treebank (PTB, Marcus et al. 1993) have a core projective tree structure and trace edges that represent control structures, wh-movement and more. However, most parsers and the standard evaluation metric ignore these edges and all null elements. By leaving out parts of the structure, they fail to provide key relations to downstream tasks such as question answering. While there has been work on capturing some parts of this extra structure, it has generally either been through post-processing on trees (Johnson 2002; Jijkoun 2003; Campbell 2004; Levy and Manning 2004; Gabbard et al. 2006) or has only captured a limited set of phenomena via grammar augmentation (Collins 1997; Dienes and Dubey 2003; Schmid 2006; Cai et al. 2011).

We propose a new general-purpose parsing algorithm that can efficiently search over a wide range of syntactic phenomena. Our algorithm extends a non-projective tree parsing algorithm (Pitler et al. 2013; Pitler 2014) to graph structures, with improvements to avoid derivational ambiguity while maintaining an $O(n^4)$ runtime. Our algorithm also includes an optional extension to ensure parses contain a directed projective tree of non-trace edges.

Our algorithm cannot apply directly to constituency parses–it requires lexicalized structures similar to dependency parses. We extend and improve previous work on lexicalized constituent representations (Shen et al. 2007; Carreras et al. 2008; Hayashi and Nagata 2016) to handle traces. In this form, traces can create problematic structures such as directed cycles, but we show how careful choice of head rules can minimize such issues.

We implement a proof-of-concept parser, scoring 88.1 on trees in section 23 and 70.6 on traces. Together, our representation and algorithm cover 97.3% of sentences, far above the coverage of projective tree parsers (43.9%).

2 Background

This work builds on two areas: non-projective tree parsing, and parsing with null elements.

Non-projectivity is important in syntax for rep-

Transactions of the Association for Computational Linguistics, vol. 5, pp. 441–454, 2017. Action Editor: Marco Kuhlmann.
Submission batch: 4/2017; Published 11/2017.

resenting many structures, but inference over the space of all non-projective graphs is intractable. Fortunately, in practice almost all parses are covered by well-defined subsets of this space. For dependency parsing, recent work has defined algorithms for inference within various subspaces (Gómez-Rodríguez and Nivre 2010; Pitler et al. 2013). We build upon these algorithms and adapt them to constituency parsing. For constituency parsing, a range of formalisms have been developed that are mildly-context sensitive, such as CCG (Steedman 2000), LFG (Kaplan and Bresnan 1982), and LTAG (Joshi and Schabes 1997).

Concurrently with this work, Cao et al. (2017) also proposed a graph version of Pitler et al. (2013)'s One-Endpoint Crossing (1-EC) algorithm. However, Cao's algorithm does not consider the direction of edges[1] and so it could produce cycles, or graphs with multiple root nodes. Their algorithm also has spurious ambiguity, with multiple derivations of the same parse structure permitted. One advantage of their algorithm is that by introducing a new item type it can handle some cases of the Locked-Chain we define below (specifically, when N is even), though in practise they also restrict their algorithm to ignore such cases. They also show that the class of graphs they generate corresponds to the 1-EC pagenumber-2 space, a property that applies to this work as well[2].

Parsing with Null Elements in the PTB has taken two general approaches. The first broadly effective system was Johnson (2002), which post-processed the output of a parser, inserting extra elements. This was effective for some types of structure, such as null complementizers, but had difficulty with long distance dependencies. The other common approach has been to thread a trace through the tree structure on the non-terminal symbols. Collins (1997)'s third model used this approach to recover wh-traces, while Cai et al. (2011) used it to recover null pronouns, and others have used it for a range of movement types (Dienes and Dubey 2003; Schmid 2006). These approaches have the disadvantage that each additional trace dramatically expands the grammar.

Our representation is similar to LTAG-Spinal (Shen et al. 2007) but has the advantage that it can be converted back into the PTB representation. Hayashi and Nagata (2016) also incorporated null elements into a spinal structure but did not include a representation of co-indexation. In related work, dependency parsers have been used to assist in constituency parsing, with varying degrees of representation design, but only for trees (Hall, Nivre, and Nilsson 2007; Hall and Nivre 2008; Fernández-González and Martins 2015; Kong et al. 2015).

Kato and Matsubara (2016) described a new approach, modifying a transition-based parser to recover null elements and traces, with strong results, but using heuristics to determine trace referents.

3 Algorithm

Our algorithm is a dynamic program, similar at a high level to CKY (Kasami 1966; Younger 1967; Cocke 1969). The states of our dynamic program (*items*) represent partial parses. Usually in CKY, items are defined as covering the n words in a sentence, starting and ending at the spaces between words. We follow Eisner (1996), defining items as covering the $n-1$ spaces in a sentence, starting and ending on words, as shown in Figure 1. This means that we process each word's left and right dependents separately, then combine the two halves.

We use three types of items: (1) a single *edge*, linking two words, (2) a continuous *span*, going from one word to another, representing all edges linking pairs of words within the span, (3) a span (as defined in 2) plus an additional word outside the span, enabling the inclusion of edges between that word and words in the span.

Within the CKY framework, the key to defining our algorithm is a set of rules that specify which items are allowed to combine. From a bottom-up perspective, a parse is built in a series of steps, which come in three types: (1) adding an edge to an item, (2) combining two items that have non-overlapping adjacent spans to produce a new item with a larger span, (3) combining three items, similarly to (2).

Example: To build intuition for the algorithm, we will describe the derivation in Figure 1. Note, item sub-types (I, X, and N) are defined below, and in-

[1] To produce directed edges, their parser treats the direction as part of the edge label.

[2] This is a topological space with two half-planes sharing a boundary. All edges are drawn on one of the two half-planes and each half-plane contains no crossings.

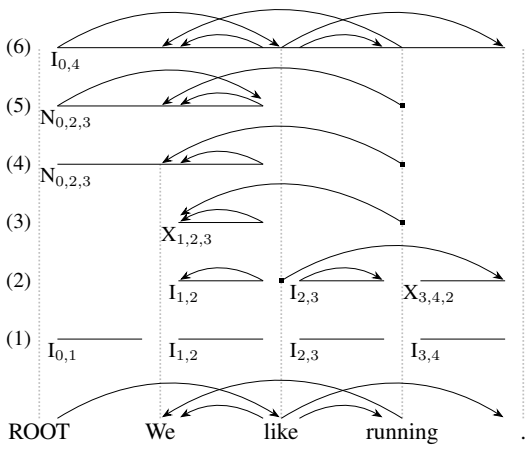


Figure 1: An example derivation using our graph parsing deduction rules.

cluded here for completeness.

(1) We initialize with spans of width one, going between adjacent words, e.g. between *ROOT* and *We*.

$$\emptyset \mapsto I_{0,1}$$

(2) Edges can be introduced in exactly two ways, either by linking the two ends of a span, e.g. *like–running*, or by linking one end of a span with a word outside the span, e.g. *like–*. (which in this case forms a new item that has a span and an external word).

$$I_{2,3} \wedge like\text{–}running \mapsto I_{2,3}$$
$$I_{3,4} \wedge like\text{–}. \mapsto X_{3,4,2}$$

(3) We add a second edge to one of the items.

$$I_{1,2} \wedge running\text{–}We \mapsto X_{1,2,3}$$

(4) Now that all the edges to *We* have been added, the two items either side of it are combined to form an item that covers it.

$$I_{0,1} \wedge X_{1,2,3} \mapsto N_{0,2,3}$$

(5) We add an edge, creating a crossing because *We* is an argument of a word to the right of *like*.

$$N_{0,2,3} \wedge ROOT\text{–}like \mapsto N_{0,2,3}$$

(7) We use a ternary rule to combine three adjacent items. In the process we create another crossing.

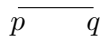
$$N_{0,2,3} \wedge I_{2,3} \wedge X_{3,4,2} \mapsto I_{0,6}$$

3.1 Algorithm definition

Notation Vertices are p, q, etc. Continuous ranges are $[pq]$, $[pq)$, $(pq]$, or (pq), where the brackets indicate inclusion, [ ], or exclusion, ( ), of each endpoint. A span $[pq]$ and vertex o that are part of the same item are $[pq.o]$. Two vertices and an arrow indicate an edge, $p\vec{q}$. Two vertices without an arrow are an edge in either direction, pq. Ranges and/or vertices connected by a dash define a set of edges, e.g. the set of edges between o and (pq) is o–(pq) (in some places we will also use this to refer to an edge from the set, rather than the whole set). If there is a path from p to q, q is *reachable* from p.

Item Types As shown in Figure 1, our items start and end on words, fully covering the spaces in between. Earlier we described three item types: an edge, a span, and a span plus an external vertex. Here we define spans more precisely as I, and divide the span plus an external point case into five types differing in the type of edge crossing they contain:

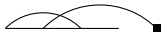
I**, Interval** A span for which there are no edges $sr : r \in (pq)$ and $s \notin [pq]$.



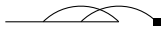
X**, Exteral** An interval and either op or oq, where $o \notin [pq]$.

B**, Both** A span and vertex $[pq.o]$, for which there are no edges $sr : r \in (pq)$ and $s \notin [pq] \cup o$. Edges o–$[pq]$ may be crossed by pq, p–(pq) or q–(pq), and at least one crossing of the second and third types occurs. Edges o–(pq) may not be crossed by (pq)–(pq) edges.

L**, Left** Same as B, but o–(pq) edges may only cross p–$(pq]$ edges.



R**, Right** Symmetric with L.



N**, Neither** An interval and a vertex $[pq.o]$, with at least one o–(pq) edge, which can be crossed by pq, but no other $[pq]$–$[pq]$ edges.

Items are further specified as described in Alg. 1. Most importantly, for each pair of o, p, and q in an item, the rules specify whether one is a parent of the other, and if they are directly linked by an edge.

For an item H with span $[ij]$, define $covered(H)$ as (ij), and define $visible(H)$ as $\{i, j\}$. When an external vertex x is present, it is in $visible(H)$. Call the union of multiple such sets $covered(F, G, H)$, and $visible(F, G, H)$.

Deduction Rules To make the deduction rules manageable, we use templates to define some constraints explicitly, and then use code to generate the rules. During rule generation, we automatically apply additional constraints to prevent rules that would leave a word in the middle of a span without a parent or that would form a cycle (proven possible below). Algorithm 1 presents the explicit constraints. Once expanded, these give rules that specify all properties for each item (general type, external vertex position

Algorithm 1 Dynamic program for Lock-Free, One-Endpoint Crossing, Directed, Acyclic graph parsing.

Adding Edges: Consider a span $[lr]$ and vertex $x \notin [lr]$.
Edges between l and r can be added to items I, N, L, R, and B (making $\hat{L}$ and $\hat{N}$ in those cases).
Edges between l and x can be added to items I (forming an X), R, and N.
Edges between r and x can be added to items I (forming an X), L, and N.
The l–r edge cannot be added after another edge, and N items cannot get both l–x and r–x edges.
Combining Items: In the rules below the following notation is used:
For this explanation items are $T[lr\ c_{rl}\ c_{lr}]$ and $T[lrx\ c_{rl}\ c_{xl}\ c_{lr}\ c_{xr}\ c_{lx}\ c_{rx}]$.
T is the type of item. Multiple letters indicate any of those types are allowed.
For the next three types of notation, if an item does not have a mark, either option is valid.
$\dot{T}$ and $\underset{\sim}{T}$ indicate the number of edges between the external vertex and the span: one or more than one respectively.
$\cdot T$ and $T\cdot$ indicate the position of the external vertex relative to the item's span (left or right respectively).
$\hat{T}$ indicates for N and L that $\forall p \in (ij) \exists rs : i \leq r < p < s \leq j$. In (11) and (12) it is optional, but true for output iff true for input.
l, r, and x: the position of the left end of the span, the right end, and the external vertex, respectively.
c_{rl}, c_{xl}, etc: connectivity of each pair of visible vertices, from the first subscript to the second. Using c_{rl} as an example, these can be . (unconstrained), d ($\vec{rl}$ must exist), p (l is reachable from r, but $\vec{rl}$ does not exist), n (l is not reachable from r), $\overline{d}$ ($= p \vee n$), $\overline{n}$ ($= d \vee p$). Note: In the generated rules every value is d, p, or n, leading to multiple rules per template below.

$$I[ij\ \ n\overline{d}] \leftarrow \max \begin{cases} \text{(Init)}\ \ j = i{+}1 \\ (1)\ \ I[i\ \ i{+}1\ \ nn]\ \ \ I[i{+}1\ \ j\ \ \overline{n}n] \\ \max_{k\in(i,j)} \begin{cases} (2)\ \ I[ik\ \ nd]\ \ \ I[kj\ \ ..] \\ (3)\ \ BLRN{\cdot}[ikj\ \ nn\overline{ddd}]\ \ \ I[kj\ \ ..] \\ \max_{l\in(k,j)} \begin{cases} (4)\ \ RN{\cdot}[ikl\ \ nn\overline{ddd}]\ \ \ I[kl\ \ ..]\ \ \ {\cdot}LNX[ljk\ \ .\overline{d}..\overline{d}.] \\ (5)\ \ BLRN{\cdot}[ikl\ \ nn\overline{ddd}]\ \ \ I[kl\ \ ..]\ \ \ I[lj\ \ ..] \end{cases} \\ \max_{l\in(i,k)} \begin{cases} (6)\ \ I[il\ \ n.]\ \ \ {\cdot}LN[lki\ \ .\overline{d}.dnn]\ \ \ {\cdot}\underset{\sim}{N}[kjl\ \ \overline{ddd}.\overline{d}.] \\ (7)\ \ RNX{\cdot}[ilk\ \ nn.\overline{d}d\overline{d}]\ \ \ I[lk\ \ ..]\ \ \ {\cdot}\underset{\sim}{LN}[kjl\ \ .\overline{d}..\overline{d}.] \end{cases} \end{cases} \end{cases}$$

$$B{\cdot}[ijx\ \ nn\overline{ddd}] \leftarrow \max_{k\in(i,j)} \begin{cases} (8)\ \ \hat{L}\hat{N}{\cdot}[ikx\ \ nn.\overline{ddd}]\ \ \ R{\cdot}[kjx\ \ ...\overline{d}.\overline{d}] \\ (9)\ \ \hat{L}\hat{N}{\cdot}[ikx\ \ nn.\overline{ddd}]\ \ \ N{\cdot}[kjx\ \ \overline{d}.d\overline{d}.\overline{d}] \\ (10)\ \ \hat{L}\hat{N}{\cdot}[ikx\ \ nn.\overline{ddd}]\ \ \ N{\cdot}[kjx\ \ d.\overline{dd}.\overline{d}] \end{cases}$$

$$\hat{L}[ijx\ \ \overline{dddddd}] \leftarrow \max_{k\in(i,j)} \begin{cases} (11)\ \ X[ikx\ \ .\overline{d}.dnn]\ \ \ {\cdot}\hat{L}\hat{N}[kji\ \ .\overline{d}.\overline{ddd}] \\ (12)\ \ X[ikx\ \ .\overline{d}.\overline{ddd}]\ \ \ {\cdot}\hat{L}\hat{N}[kji\ \ .\overline{d}.\overline{ddd}] \end{cases}$$

$$\underset{\sim}{L}[ijx\ \ \overline{dddddd}] \leftarrow \max_{k\in(i,j)} \begin{cases} (13)\ \ LN[ikx\ \ .\overline{d}.d\overline{ddd}]\ \ \ {\cdot}N[kji\ \ \overline{dddddd}] \\ (14)\ \ LN[ikx\ \ .\overline{d}.\overline{ddd}]\ \ \ {\cdot}N[kji\ \ \overline{dddddd}] \\ (15)\ \ L[ikx\ \ .\overline{d}.d\overline{ddd}]\ \ \ I[kj\ \ ..] \\ (16)\ \ L[ikx\ \ .\overline{d}.\overline{ddd}]\ \ \ I[kj\ \ ..] \\ (17)\ \ N[ikx\ \ \overline{ddddd}]\ \ \ I[kj\ \ ..] \\ (18)\ \ N[ikx\ \ \overline{dd}\overline{ddd}]\ \ \ I[kj\ \ ..] \\ (19)\ \ N[ikx\ \ d\overline{ddddd}]\ \ \ I[kj\ \ ..] \\ (20)\ \ N[ikx\ \ d\overline{ddddd}]\ \ \ I[kj\ \ ..] \end{cases}$$

$$\underset{\sim}{N}[ijx\ \ \overline{dddddd}] \leftarrow \max_{k\in(i,j)} \begin{cases} (21)\ \ {\cdot}N[ikx\ \ \overline{ddd}\overline{ddd}]\ \ \ I[kj\ \ ..] \\ (22)\ \ {\cdot}N[ikx\ \ \overline{dddddd}]\ \ \ I[kj\ \ ..] \\ (23)\ \ I[ik\ \ ..]\ \ \ N{\cdot}[kjx\ \ \overline{dddddd}] \\ (24)\ \ I[ik\ \ ..]\ \ \ N{\cdot}[kjx\ \ \overline{dddddd}] \end{cases}$$

$$\dot{N}[ijx\ \ \overline{dddddd}] \leftarrow \max_{k\in(i,j)} \begin{cases} (25)\ \ {\cdot}X[ikx\ \ .\overline{d}.d\overline{ddd}]\ \ \ I[kj\ \ ..] \\ (26)\ \ {\cdot}X[ikx\ \ .\overline{d}.\overline{ddd}]\ \ \ I[kj\ \ ..] \\ (27)\ \ I[ik\ \ ..]\ \ \ X{\cdot}[kjx\ \ .d.\overline{ddd}] \\ (28)\ \ I[ik\ \ ..]\ \ \ X{\cdot}[kjx\ \ .\overline{d}.\overline{ddd}] \end{cases}$$

$I[ij\ \ pn]$, $\cdot B[ijx\ \ \overline{dd}nn\overline{dd}]$, $\underset{\sim}{R}[ijx\ \ \overline{dddddd}]$, and $\dot{R}[ijx\ \ \overline{dddddd}]$ are symmetric with cases above.

relative to the item spans, connectivity of every pair of vertices in each item, etc).

The final item for n vertices is an interval where the left end has a parent. For parsing we assume there is a special root word at the end of the sentence.

3.2 Properties

Definition 1. *A graph is **One-Endpoint Crossing** if, when drawn with vertices along the edge of a half-plane and edges drawn in the open half-plane above, for any edge e, all edges that cross e share a vertex. Let that vertex be $\mathcal{P}t(e)$.*

Aside from applying to graphs, this is the same as Pitler et al. (2013)'s 1-EC tree definition.

Definition 2. *A **Locked-Chain** (shown in Fig. 2) is formed by a set of consecutive vertices in order from 0 to N, where $N > 3$, with edges $\{(0, N{-}1), (1, N)\} \cup \{(i, i{+}2) \forall i \in [0, N{-}2]\}$.*

Definition 3. *A graph is **Lock-Free** if it does not contain edges that form a Locked-Chain.*

Note that in practice, most parse structures satisfy 1-EC, and the Locked-Chain structure does not occur in the PTB when using our head rules.

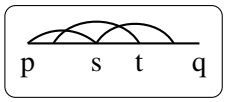
Theorem 1. *For the space of Lock-Free One-Endpoint Crossing graphs, the algorithm is sound, complete and gives unique decompositions.*

Our proof is very similar in style and structure to Pitler et al. (2013). The general approach is to consider the set of structures an item could represent, and divide them into cases based on properties of the internal structure. We then show how each case can be decomposed into items, taking care to ensure all the properties that defined the case are satisfied. Uniqueness follows from having no ambiguity in how a given structure could be decomposed. Completeness and soundness follow from the fact that our rules apply equally well in either direction, and so our top-down decomposition implies a bottom-up formation. To give intuition for the proof, we show the derivation of one rule below. The complete proof can be found in Kummerfeld (2016). We do not include it here due to lack of space.

We do provide the complete set of rule templates in Algorithm 1, and in the proof of Lemma 2 we show that the case in which an item cannot be decomposed occurs if and only if the graph contains a Locked-Chain. To empirically check our rule generation code, we checked that our parser uniquely decomposes all 1-EC parses in the PTB and is unable to decompose the rest.

Note that by using subsets of our rules, we can restrict the space of structures we generate, giving parsing algorithms for projective DAGs, projective trees (Eisner 1996), or 1-EC trees (Pitler et al. 2013). Versions of these spaces with undirected edges could also be easily handled with the same approach.

Derivation of rule (4) in Algorithm 1: This rule applies to intervals with the sub-structure shown, and with no parent in this item for p. They have at least one p–(pq) edge (otherwise rule 1 applies). The longest p–(pq) edge, ps, is crossed (otherwise rule 2 applies). Let C be the set of (ps)–(sq) edges (note: these cross ps). Either all of the edges in C have a common endpoint $t \in (sq)$, or if $|C| = 1$ let t be the endpoint in (sq) (otherwise rule 6 or 7 applies). Let D be the set of s–(tq) edges. $|D| > 0$ (otherwise rule 3 or 5 applies).



We will break this into three items. First, (st)–$(tq]$ edges would violate the 1-EC property and (st)–$[ps)$ edges do not exist by construction. Therefore, the middle item is an Interval $[st]$, the left item is $[ps.t]$, and the right item is $[tq.s]$ (since $|C| > 0$ and $|D| > 0$). The left item can be either an N or R, but not an L or B because that would violate the 1-EC property for the C edges. The right item can be an X, L, or N, but not an R or B because that would violate the 1-EC property for the D edges. We will require edge ps to be present in the first item, and not allow pt. To avoid a spurious ambiguity, we also prevent the first or third items from having st (which could otherwise occur in any of the three items). Now we have broken down the original item into valid sub-items, and we have ensured that those sub-items contain all of the structure used to define the case in a unique way.

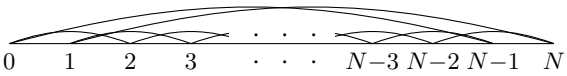


Figure 2: Visualization of Locked-Chain structures. Note, the use of 0 to N does not imply this must span the entire sentence, these numbers are just for convenience in the definition.

Now we will further characterize the nature of the Lock-Free restriction to the space of graphs.

Lemma 1. *No edge in a Locked-Chain in a 1-EC graph is crossed by edges that are not part of it.*

Proof. First, note that: $\mathcal{P}t((0, N{-}1)) = N$, $\mathcal{P}t((1, N)) = 0$, and $\{\mathcal{P}t((i, i{+}2)) = i{+}1 \ \forall i \in [0, N{-}2]\}$ Call the set $\{(i, i{+}2) \forall i \in [0, N{-}2]\}$, the *chain*.

Consider an edge e that crosses an edge f in a Locked-Chain. Let e_{in} be the end of e that is between the two ends of f, and e_{out} be the other end. One of e's endpoints is at $\mathcal{P}t(f)$, and $\mathcal{P}t(e)$ is an endpoint of f. There are three cases:

(i) $f = (1, N)$. Here, $e_{out} = \mathcal{P}t(f) = 0$, and $e_{in} \in (1, N)$. For all vertices $v \in (1, N)$ there is an edge g in the chain such that v is between the endpoints of g. Therefore, e will cross such an edge g. To satisfy the 1-EC property, g must share an endpoint with f, which means g is either $(1, 3)$ or $(N{-}2, N)$. In the first case, the 1-EC property forces $e = (0, 2)$, and in the second $e = (0, N{-}1)$, both of which are part of the Locked-Chain.

(ii) $f = (0, N{-}1)$, symmetrical with (i).

(iii) $f = (i, i{+}2)$, for some $i \in [0, N{-}2]$. Here, $e_{in} = \mathcal{P}t(f) = i{+}1$. We can assume e does not cross $(0, N{-}1)$ or $(1, N)$, as those cases are covered by (i). As in (i), e must cross another edge in the chain, and that edge must share an endpoint with f.

This forces e to be either $(i{-}1, i{+}1)$ or $(i{+}1, i{+}3)$ (excluding one or both if they cross $(0, N{-}1)$ or $(1, N)$), which are both in the Locked-Chain. □

Our rules define a unique way to decompose almost any item into a set of other items. The exception is B, which in some cases can not be divided into two items (i.e. has no valid binary division).

Lemma 2. *A $B[ij.x]$ has no valid binary division if and only if the graph has a Locked-Chain.*

Proof. Consider the k and l that give the longest ik and lj edges in a B with no valid binary division (at least one edge of each type must exist by definition). No vertex in (ik) or (jl) is a valid split point, as they would all require one of the items to have two external vertices.

Now, consider $p \in [kj]$. If there is no edge $l_1 r_1$, where $i \leq l_1 < p < r_1 \leq j$, then p would be a valid split point. Therefore, such an edge must exist. Consider l_1, either $l_1 \in (ik)$ or there is an edge $l_2 c$, where $i \leq l_2 < l_1 < c \leq j$ (by the same logic as for $l_1 r_1$). Similarly, either $r_1 \in (jl)$ or there is an edge $c r_2$ (it must be c to satisfy 1-EC). We can also apply this logic to edges $l_2 c$ and $c r_2$, giving edges $l_3 l_1$ and $r_1 r_3$. This pattern will terminate when it reaches $l_u \in (ik)$ and $r_v \in (jl)$ with edges $l_u l_{u-2}$ and $r_{v-2} r_v$. Note that $k = l_{u-1}$ and $l = r_{v-1}$, to satisfy 1-EC.

Since it is a B, there must be at least two x–(ij) edges. To satisfy 1-EC, these end at l_{u-1} and r_{v-1}.

Let x be to the right (the left is symmetrical), and call $i = 0$, $j = N{-}1$, and $x = N$. Comparing with the Locked-Chain definition, we have all the edges except one: 0 to $N{-}1$. However, that edge must be present in the overall graph, as all B items start with an ij edge (see rules 3 and 5 in Algorithm 1). Therefore, if there is no valid split point for a B, the overall graph must contain a Locked-Chain.

Now, for a graph that contains a Locked-Chain, consider the items that contain the Locked-Chain. Grouping them by their span $[ij]$, there are five valid options: $[0, N{-}1]$, $[1, N]$, $[0, N]$, $(i \leq 0 \wedge j > N)$, and $(i < 0 \wedge j \geq N)$. Items of the last three types would be divided by our rules into smaller items, one of which contains the whole Locked-Chain. The first two are Bs of the type discussed above. □

Now we will prove that our code to generate rules from the templates can guarantee a DAG is formed.

Lemma 3. *For any item H, $\forall v \in covered(H)$ $\exists u \in visible(H) : v$ is reachable from u.*

Proof. This is true for initial items because $covered(H) = \emptyset$. To apply induction, consider adding edges and combing items. The lemma clearly remains true when adding an edge. Consider combining items E, F, G to form $H[ij.x]$, and assume the lemma is true for E, F, and G (the binary case is similar). Since all vertices are reachable from $visible(E, F, G)$, we only need to ensure that $\forall v \in visible(E, F, G)$ $\exists u \in visible(H) : v$ is reachable from u. The connectivity between all pairs $\{(u, v) \mid u \in visible(H), v \in visible(E, F, G)\}$ can be inferred from the item definitions, and so this requirement can be enforced in rule generation. □

Lemma 4. *The final item is a directed acyclic graph.*

Proof. First, consider acyclicity. Initial items do not contain any edges and so cannot contain a cycle. For induction, there are two cases:

(i) Adding an Edge $\vec{pq}$ to an item H: Assume that H does not contain any cycles. $\vec{pq}$ will create a cycle if and only if p is reachable from q. By construction, p and $q \in visible(H)$, and so the item definition contains whether p is reachable from q.

(ii) Combining Items: Assume that in isolation, none of the items being combined contain cycles. Therefore, a cycle in the combined item must be composed of paths in multiple items. A path in one item can only continue in another item by passing through a visible vertex. Therefore, a cycle would have to be formed by a set of paths between visible vertices. But the connectivity of every pair of visible vertices is specified in the item definitions.

In both cases, rules that create a cycle can be excluded during rule generation.

By induction, the items constructed by our algorithm do not contain cycles. Together with Lemma 3 and the final item definition, this means the final structure is an acyclic graph with all vertices reachable from vertex n. □

Next, we will show two properties that give intuition for the algorithm. Specifically, we will prove which rules add edges that are crossed in the final derivation.

Lemma 5. *An edge ij added to $I[ij]$ is not crossed.*

Proof. First, we will show three properties of any pair of items in a derivation (using $[ij.x]$ and $[kl.y]$).

(1) *It is impossible for either* $i < k < j < l$ *or* $k < i < l < j$, i.e., items cannot have partially overlapping spans. As a base case, the final item is an interval spanning all vertices, and so no other item can partially overlap with it. Now assume it is true for an item H and consider the rules in reverse, breaking H up. By construction, each rule divides H into items with spans that are adjacent, overlapping only at their visible vertices. Since the new items are nested within H, they do not overlap with any items H did not overlap with. By induction, no pair of items have partially overlapping spans.

(2) *For items with nested spans* ($i \leq k < l \leq j$), $y \in [ij] \cup \{x\}$. Following the argument for the previous case, the $[ij.x]$ item must be decomposed into a set of items that includes $[kl.y]$. Now, consider how those items are combined. The rules that start with an item with an external vertex produce an item that either has the same external vertex, or with the external vertex inside the span of the new item. Therefore, y must either be equal to x or inside $[ij]$.

(3) *For items without nested spans,* $x \notin (kl)$. Assume $x \in (kl)$ for two items without nested spans. None of the rules combine such a pair of items, or allow one to be extended so that the other is nested within it. However, all items are eventually combined to complete the derivation. By contradiction, $x \notin (kl)$.

Together, these mean that given an interval H with span $[ij]$, and another item G, either $\forall v \in visible(G), v \in [ij]$ or $\forall v \in visible(G), v \notin (ij)$. Since edges are only created between visible vertices, no edge can cross edge ij. □

Lemma 6. *All edges aside from those considered in Lemma 5 are crossed.*

Proof. First, consider an edge ij added to an item $[ij.x]$ of type B, L, R, or N. This edge is crossed by all x–(ij) edges, and in these items $|x\text{–}(ij)| \geq 1$ by definition. Note, by the same argument as Lemma 5, the edge is not crossed later in the derivation.

Second, consider adding $e \in \{xi, xj\}$, to H, an item with $[ij]$ or $[ij.x]$, forming an item $G[ij.x]$. Note, e does not cross any edges in H. Let $E(F[kl.y])$ be the set of y–$[kl]$ edges in some item F. Note that $e \in E(G)$. We will show how this set of edges is affected by the rules and what that implies for e. Consider each input item $A[kl.y]$ in each rule, with output item C. Every item A falls into one of four categories: (1) $\forall f \in E(A)$, f is crossed by an edge in another of the rule's input items, (2) $E(A) \subseteq E(C)$, (3) $A \wedge kl \mapsto C$ and there are no ky or ly edges in A, (4) A contains edge kl and there are no ky or ly edges in A.

Cases 2-4 are straightforward to identify. For an example of the first case, consider the rightmost item in rule 4. The relevant edges are k–$(lj]$ (by construction, kl is not present). Since the leftmost item is either an R or N, $|l\text{–}(ik)| \geq 1$. Since $i < k < l < j$, all k–$(lj]$ edges will cross all l–$[ik)$ edges. Therefore applying this rule will cross all k–$(lj]$ edges in the rightmost item.

Initially, e is not crossed and $e \in E(G)$. For each rule application, edges in $E(A)$ are either crossed (1 and 3), remain in the set $E(C)$ (2), or must already be crossed (4). Since the final item is an interval and $E(\text{Interval}) = \emptyset$, there must be a subsequent rule that is not in case 2. Therefore e will be crossed. □

3.3 Comparison with Pitler et al. (2013)

Our algorithm is based on Pitler et al. (2013), which had the crucial idea of One-Endpoint crossing and a complete decomposition of the tree case. Our changes and extensions provide several benefits:

Extension to graphs: By extending to support multiple parents while preventing cycles, we substantially expand the space of generatable structures.

Uniqueness: By avoiding derivational ambiguity we reduce the search space and enable efficient summing as well as maxing. Most of the cases in which ambiguity arises in Pitler et al. (2013)'s algorithm are due to symmetry that is not explicitly broken. For example, the rule we worked through in the previous section defined $t \in (sq)$ when $|C| = 1$. Picking $t \in (ps)$ would also lead to a valid set of rules, but allowing either creates a spurious ambiguity. This ambiguity is resolved by tracking whether there is only one edge to the external vertex or more than one, and requiring more than one in rules 6 and 7. Other changes include ensuring equivalent structures cannot be represented by multiple item types and enforcing a unique split point in B items.

More concise algorithm definition: By separating edge creation from item merging, and defining our rules via a combination of templates and code, we are able to define our algorithm more concisely.

3.4 Algorithm Extensions

3.4.1 Edge Labels and Word Labels

Edge labels can be added by calculating either the sum or max over edge types when adding each edge. Word labels (e.g., POS Tags) must be added to the state, specifying a label for each visible word (p, q and o). This state expansion is necessary to ensure agreement when combining items.

3.4.2 Ensuring a Structural Tree is Present

Our algorithm constrains the space of graph structures, but we also want to ensure that our parse contains a projective tree of non-trace edges.

To ensure every word gets one and only one structural parent, we add booleans to the state, indicating whether p, q and o have structural parents. When adding edges, a structural edge cannot be added if a word already has a structural parent. When combining items, no word can receive more than one structural parent, and words that will end up in the middle of the span must have exactly one. Together, these constraints ensure we have a tree.

To ensure the tree is projective, we need to prevent structural edges from crossing. Crossing edges are introduced in two ways, and in both we can avoid structural edges crossing by tracking whether there are structural o–$[pq]$ edges. Such edges are present if a rule adds a structural op or oq edge, or if a rule combines an item with structural o–$[pq]$ edges and o will still be external in the item formed by the rule.

For adding edges, every time we add a pq edge in the N, L, R and B items we create a crossing with all o–(pq) edges. We do not create a crossing with oq or op, but our ordering of edge creation means these are not present when we add a pq edge, so tracking structural o–$[pq]$ edges gives us the information we need to prevent two structural edges crossing.

For combining items, in Lemma 6 we showed that during combinations, o–$[pq]$ edges in each pair of items will cross. As a result, knowing whether any o–$[pq]$ edge is structural is sufficient to determine whether two structural edges will cross.

3.5 Complexity

Consider a sentence with n tokens, and let E and S be the number of edge types and word labels in our grammar respectively.

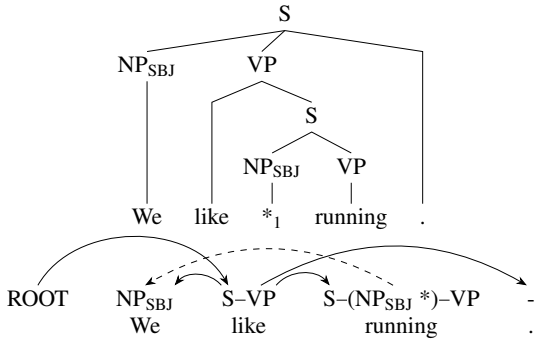


Figure 3: Parse representations for graph structures, PTB (top) and ours (bottom).

Parses without word or edge labels: Rules have up to four positions, leading to complexity of $O(n^4)$. Note, there is also an important constant–once our templates are expanded, there are 49,292 rules.

With edge labels: When using a first-order model, edge labels only impact the rules for edge creation, leading to a complexity of $O(n^4 + En^2)$.

With word labels: Since we need to track word labels in the state, we need to adjust every n by a factor of S, leading to $O(S^4n^4 + ES^2n^2)$.

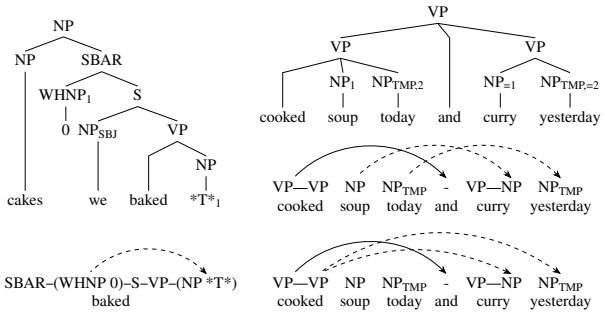
4 Parse Representation

Our algorithm relies on the assumption that we can process the dependents to the left and right of a word independently and then combine the two halves. This means we need lexicalized structures, which the PTB does not provide. We define a new representation in which each non-terminal symbol is associated with a specific word (the head). Unlike dependency parsing, we retain all the information required to reconstruct the constituency parse.

Our approach is related to Carreras et al. (2008) and Hayashi and Nagata (2016), with three key differences: (1) we encode non-terminals explicitly, rather than implicitly through adjunction operations, which can cause ambiguity, (2) we add representations of null elements and co-indexation, (3) we modify head rules to avoid problematic structures.

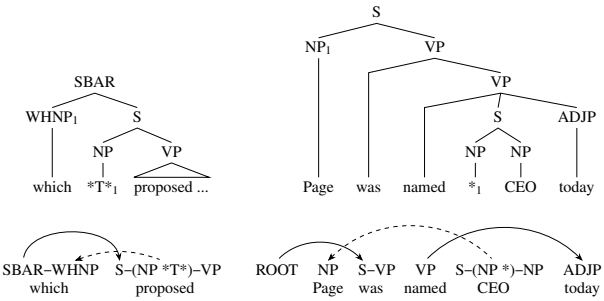
Figure 3 shows a comparison of the PTB representation and ours. We add lexicalization, assigning each non-terminal to a word. The only other changes are visual notation, with non-terminals moved to be directly above the words to more clearly show the distinction between *spines* and *edges*.

Spines: Each word is assigned a spine, shown im-



(a) Null to null (b) Parallel Constructions

Figure 4: Examples of syntactic phenomena. Only relevant edges and spines are shown.



(a) Cycle (b) Not One-Endpoint Crossing

Figure 5: Examples of problematic graph structured syntactic phenomena before our head rule changes.

mediately above the word. A spine is the ordered set of non-terminals that the word is the head of, e.g. S-VP for *like*. If a symbol occurs more than once in a spine, we use indices to distinguish instances.

Edges: An edge is a link between two words, with a label indicating the symbols it links in the child and parent spines. In our figures, edge labels are indicated by where edges start and end.

Null Elements: We include each null element in the spine of its parent, unlike Hayashi and Nagata (2016), who effectively treated null elements as words, assigning them independent spines. We also considered encoding null elements entirely on edges but found this led to poorer performance.

Co-indexation: The treebank represents movement with index pairs on null elements and non-terminals, e.g. $*_1$ and NP_1 in Figure 3. We represent co-indexation with edges, one per reference, going from the null element to the non-terminal. There are three special cases of co-indexation:

(1) It is possible for trace edges to have the same start and end points as a non-trace edge. We restrict this case to allow at most one trace edge. This decreases edge coverage in the training set by 0.006%.

(2) In some cases the reference non-terminal only spans a null element, e.g. the WHNP in Figure 4a. For these we use a reversed edge to avoid creating a cycle. Figure 4a shows a situation where the trace edge links two positions in the same spine, which we assign with the spine during parsing.

(3) For parallel constructions the treebank co-indexes arguments that fulfill the same roles (Fig. 4b). These are distinct from the previous cases because neither index is on a null element. We considered two options: add edges from the repetition to the referent (middle), or add edges from the repetition to the parent of the first occurrence (bottom). Option two produces fewer non-1-EC structures and explicitly represents all predicates, but only implicitly captures the original structure.

4.1 Avoiding Adjunction Ambiguity

Prior work on parsing with spines has used r-adjunction to add additional non-terminals to spines. This introduces ambiguity, because edges modifying the same spine from different sides may not have a unique order of application. We resolve this issue by using more articulated spines with the complete set of non-terminals. We found that 0.045% of spine instances in the development set are not observed in training, though in 70% of those cases an equivalent spine sans null elements is observed in training.

4.2 Head Rules

To construct the spines, we lexicalize with head rules that consider the type of each non-terminal and its children. Different heads often represent more syntactic or semantic aspects of the phrase. For trees, all head rules generate valid structures. For graphs, head rules influence the creation of two problematic structures:

Cycles: These arise when the head chosen for a phrase is also an argument of another word in the phrase. Figure 5a shows a cycle between *which* and *proposed*. We resolve this by changing the head of an SBAR to be an S rather than a Wh-noun phrase.

One-Endpoint Crossing Violations: Figure 5b shows an example, with the trace from *CEO* to *Page* crossing two edges with no endpoints in common. We resolve this case by changing the head for VPs to be a child VP rather than an auxiliary.

5 Results

Algorithm Coverage: In Table 1 we show the impact of design decisions for our representation. The percentages indicate how many sentences in the training set are completely recoverable by our algorithm. Each row shows the outcome of an addition to the previous row, starting from no traces at all, going to our representation with the head rules of Carreras et al. (2008), then changing the head rules, reversing null-null edges, and changing the target of edges in parallel constructions. The largest gain comes from changing the head rules, which is unsurprising since Carreras et al. (2008)'s rules were designed for trees (any set of rules form valid structures for trees).

Problematic Structures: Of the sentences we do not cover, 54% contain a cycle, 45% contain a 1-EC violation, and 1% contain both. To understand these problematic sentences, we manually inspected a random sample of twenty parses that contained a cycle and twenty parses with a 1-EC violation (these forty are 6% of all problematic parses, enough to identify the key remaining challenges).

For the cycles, eleven cases related to sentences containing variations of NP *said* interposed between two parts of a single quote. A cycle was present because the top node of the parse was co-indexed with a null argument of *said* while *said* was an argument of the head word of the quote. The remaining cases were all instances of pseudo-attachment, which the treebank uses to show that non-adjacent constituents are related (Bies et al. 1995). These cases were split between use of Expletive (5) and Interpret Constituent Here (4) traces.

It was more difficult to determine trends for cases where the parse structure has a 1-EC violation. The same three cases, Expletive, Interpret Constituent Here, and NP *said* accounted for half of the issues.

5.1 Implementation

We implemented a parser with a first-order model using our algorithm and representation. Code for the parser, for conversion to and from our representation, and for our metrics is available[3]. Our parser uses a linear discriminative model, with features based on McDonald et al. (2005). We train with an online primal subgradient approach (Ratliff et al. 2007) as described by Kummerfeld, Berg-Kirkpatrick, et al. (2015), with parallel lock-free sparse updates.

Representation	Coverage (%) Sentences	Edges
Projective trees, no nulls	26.59	96.27
Projective trees, with nulls	43.85	96.27
Projective graphs	50.60	96.67
One-EC graphs	71.84	98.31
+ Head rule changes	92.35	99.23
+ Null reversal	97.02	99.45
+ Parallel construction shift	97.31	99.49

Table 1: Training set coverage for different representations. One-EC graphs uses our representation, but with the head rules from Carreras et al. (2008). For the edge results, we only exclude edges necessary to make each parse representable (e.g. excluding only one edge in a cycle and counting the rest).

Loss Function: We use a weighted Hamming distance for loss-augmented decoding, as it can be efficiently decomposed within our dynamic program. Calculating the loss for incorrect spines and extra edges is easy. For missing edges, we add when a deduction rule joins two spans that cover an end of the edge, since if it does not exist in one of those items it is not going to be created in future. To avoid double counting we subtract when combining two halves that contain the two ends of a gold edge[4].

Inside–Outside Calculations: Assigning scores to edges is simple, as they are introduced in a single item in the derivation. Spines must be introduced in multiple items (left, right, and external positions) and must be assigned a score in every case to avoid ties in beams. We add the score every time the spine is introduced and then subtract when two items with a spine in common are combined.

Algorithm rule pruning: Many 1-EC structures are not seen in our data. We keep only the rules used in gold training parses, reducing the set of 49,292 from the general algorithm to 627 (including rules for both adding arcs and combining items). Almost every template in Algorithm 1 generates some unnecessary rules, and no items of type B are needed.

[3] https://github.com/jkkummerfeld/1ec-graph-parser

[4] One alternative is to count half of it on each end, removing the need for subtraction later. Another is to add it during the combination step.

The remaining rules still have high coverage of the development set, missing only 15 rules, each applied once (out of 78,692 rule applications). By pruning in this way, we are considering the intersection of 1-EC graphs and the true space of structures used in language.

Chart Pruning: To improve speed we use beams and cube pruning (Chiang 2007), discarding items based on their Viterbi inside score. We divide each beam into sub-beams based on aspects of the state. This ensures diversity and enables consideration of only compatible items during binary and ternary compositions.

Coarse to Fine Pruning: Rather than parsing immediately with the full model we use several passes with progressively richer structure (Goodman 1997): (1) Projective parsing without traces or spines, and simultaneously a trace classifier, (2) Non-projective parsing without spines, and simultaneously a spine classifier, (3) Full structure parsing. Each pass prunes using parse max-marginals and classifier scores, tuned on the development set. The third pass also prunes spines that are not consistent with any unpruned edge from the second pass. For the spine classifier we use a bidirectional LSTM tagger, implemented in DyNet (Neubig et al. 2017).

Speed: Parsing took an average of 8.6 seconds per sentence for graph parsing and 0.5 seconds when the parser is restricted to trees[5]. Our algorithm is also amenable to methods such as semi-supervised and adaptive supertagging, which can improve the speed of a parser after training (Kummerfeld, Roesner, et al. 2010; Lewis and Steedman 2014).

Tree Accuracy: On the standard tree-metric, we score 88.1. Using the same non-gold POS tags as input, Carreras et al. (2008) score 90.9, probably due to their second-order features and head rules tuned for performance[6]. Shifting to use their head rules, we score 88.9. Second-order features could be added to our model through the use of forest reranking, an improvement that would be orthogonal to this paper's contributions.

We can also evaluate on spines and edges. Since their system produces regular PTB trees, we convert its output to our representation and compare its results with our system using their head rules. We see slightly lower accuracy for our system on both spines (94.0 vs. 94.3) and edges (90.4 vs. 91.1).

System	P	R	F
Null Elements Only			
Johnson (2002)	85	74	79
Hayashi and Nagata (2016)	90.3	81.7	85.8
Kato and Matsubara (2016)	88.5	82.1	85.2
This work	89.5	81.6	85.4
Null Elements and Co-indexation			
Johnson (2002)	73	63	68
Kato and Matsubara (2016)	81.2	74.7	77.8
This work	74.3	67.3	70.6

Table 2: Accuracy on section 23 using Johnson's metric.

Trace Accuracy: Table 2 shows results using Johnson (2002)'s trace metric. Our parser is competitive with previous work that has highly-engineered models: Johnson's system has complex non-local features on tree fragments, and similarly Kato and Matsubara (K&M 2016) consider complete items in the stack of their transition-based parser. On co-indexation our results fall between Johnson and K&M. Converting to our representation, our parser has higher precision than K&M on trace edges (84.1 vs. 78.1) but lower recall (59.5 vs. 71.3). One modeling challenge we observed is class imbalance: of the many places a trace could be added, only a small number are correct, and so our model tends to be conservative (as shown by the P/R tradeoff).

6 Conclusion

We propose a representation and algorithm that cover 97.3% of graph structures in the PTB. Our algorithm is $O(n^4)$, uniquely decomposes parses, and enforces the property that parses are composed of a core tree with additional traces and null elements. A proof of concept parser shows that our algorithm can be used to parse and recover traces.

Acknowledgments

We thank Greg Durrett for advice on parser implementation and debugging, and the action editor and anonymous reviewers for their helpful feedback. This research was partially supported by a General Sir John Monash Fellowship and the Office of Naval

[5] Using a single core of an Amazon EC2 m4.2xlarge instance (2.4 GHz Xeon CPU and 32 Gb of RAM).

[6] Previous work has shown that the choice of head can significantly impact accuracy (Schwartz et al. 2012).

Research under MURI Grant No. N000140911081.

References

Ann Bies, Mark Ferguson, Karen Katz, Robert MacIntyre, Victoria Tredinnick, Grace Kim, Mary Ann Marcinkiewicz, and Britta Schasberger (1995). *Bracketing Guidelines for Treebank 2 Style Penn Treebank Project*. Tech. rep.

Shu Cai, David Chiang, and Yoav Goldberg (2011). Language-Independent Parsing with Empty Elements. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. https://www.aclweb.org/anthology/P/P11/P11-2037.pdf.

Richard Campbell (2004). Using Linguistic Principles to Recover Empty Categories. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics (ACL)*. https://www.aclweb.org/anthology/P/P04/P04-1082.pdf.

Junjie Cao, Sheng Huang, Weiwei Sun, and Xiaojun Wan (2017). Parsing to 1-Endpoint-Crossing, Pagenumber-2 Graphs. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Xavier Carreras, Michael Collins, and Terry Koo (2008). TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-rich Parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning (CoNLL)*. https://www.aclweb.org/anthology/W/W08/W08-2102.pdf.

David Chiang (2007). Hierarchical Phrase-Based Translation. *Computational Linguistics* 33.2, pp. 201–228. https://www.aclweb.org/anthology/J/J07/J07-2003.pdf.

Noam Chomsky (1981). *Lectures on government and binding: The Pisa lectures*.

John Cocke (1969). *Programming Languages and Their Compilers: Preliminary Notes*. Tech. rep. Courant Institute of Mathematical Sciences, New York University.

Michael Collins (1997). Three Generative, Lexicalised Models for Statistical Parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL)*. https://www.aclweb.org/anthology/P/P97/P97-1003.pdf.

Pétr Dienes and Amit Dubey (2003). Deep Syntactic Processing by Combining Shallow Methods. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*. https://www.aclweb.org/anthology/P/P03/P03-1055.pdf.

Jason Eisner (1996). Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (CoLing)*. http://www.aclweb.org/anthology/C/C96/C96-1058.pdf.

Daniel Fernández-González and André F. T. Martins (2015). Parsing as Reduction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*. https://www.aclweb.org/anthology/P/P15/P15-1147.pdf.

Ryan Gabbard, Mitchell Marcus, and Seth Kulick (2006). Fully Parsing the Penn Treebank. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the ACL (NAACL-HLT)*. https://www.aclweb.org/anthology/N/N06/N06-1024.pdf.

Carlos Gómez-Rodríguez and Joakim Nivre (2010). A Transition-based Parser for 2-Planar Dependency Structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*. https://www.aclweb.org/anthology/P/P10/P10-1151.pdf.

Joshua Goodman (1997). Global Thresholding and Multiple-Pass Parsing. In *Processings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://www.aclweb.org/anthology/W/W97/W97-0302.pdf.

Johan Hall and Joakim Nivre (2008). A Dependency-Driven Parser for German Dependency and Constituency Representations. In *Proceedings of the Workshop on Parsing German*. https://www.aclweb.org/anthology/W/W08/W08-1007.pdf.

Johan Hall, Joakim Nivre, and Jens Nilsson (2007). A Hybrid Constituency-Dependency Parser for Swedish. In *Proceedings of the 16th Nordic Conference of Computational Linguistics (NODALIDA)*. https://www.aclweb.org/anthology/W/W07/W07-2444.pdf.

Katsuhiko Hayashi and Masaaki Nagata (2016). Empty Element Recovery by Spinal Parser Operations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL, Volume 2: Short Papers)*. https://www.aclweb.org/anthology/P/P16/P16-2016.pdf.

Valentin Jijkoun (2003). Finding Non-local Dependencies: Beyond Pattern Matching. In *The Companion*

Volume to the Proceedings of 41st Annual Meeting of the Association for Computational Linguistics (ACL-SRW). https://www.aclweb.org/anthology/P/P03/P03-2006.pdf.

Mark Johnson (2002). A Simple Pattern-matching Algorithm for Recovering Empty Nodes and Their Antecedents. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. https://www.aclweb.org/anthology/P/P02/P02-1018.pdf.

Aravind K. Joshi and Yves Schabes (1997). Handbook of Formal Languages: Volume 3 Beyond Words. In. Springer Berlin Heidelberg. Chap. Tree-Adjoining Grammars, pp. 69–123.

R. M. Kaplan and J. Bresnan (1982). Lexical-Functional Grammar: A Formal System for Grammatical Representation. In *The Mental Representation of Grammatical Relations*. MIT Press, pp. 173–281.

Tadao Kasami (1966). *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Tech. rep. University of Illinois at Urbana-Champaign.

Yoshihide Kato and Shigeki Matsubara (2016). Transition-Based Left-Corner Parsing for Identifying PTB-Style Nonlocal Dependencies. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL, Volume 1: Long Papers)*. http://www.aclweb.org/anthology/P/P16/P16-1088.pdf.

Lingpeng Kong, Alexander M. Rush, and Noah A. Smith (2015). Transforming Dependencies into Phrase Structures. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. https://www.aclweb.org/anthology/N/N15/N15-1080.pdf.

Jonathan K. Kummerfeld (2016). *Algorithms for Identifying Syntactic Errors and Parsing with Graph Structured Output*. PhD thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-138.html.

Jonathan K. Kummerfeld, Taylor Berg-Kirkpatrick, and Dan Klein (2015). An Empirical Analysis of Optimization for Max-Margin NLP. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://www.aclweb.org/anthology/D/D15/D15-1032.pdf.

Jonathan K. Kummerfeld, Jessica Roesner, Tim Dawborn, James Haggerty, James R. Curran, and Stephen Clark (2010). Faster Parsing by Supertagger Adaptation. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*. http://www.aclweb.org/anthology/P/P10/P10-1036.pdf.

Roger Levy and Christopher D. Manning (2004). Deep Dependencies from Context-free Statistical Parsers: Correcting the Surface Dependency Approximation. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*. https://www.aclweb.org/anthology/P/P04/P04-1042.pdf.

Mike Lewis and Mark Steedman (2014). Improved CCG Parsing with Semi-supervised Supertagging. *Transactions of the Association for Computational Linguistics* 2, pp. 327–338. ISSN: 2307-387X. https://transacl.org/ojs/index.php/tacl/article/view/388.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics* 19.2, pp. 313–330. https://www.aclweb.org/anthology/J/J93/J93-2004.pdf.

Ryan McDonald, Koby Crammer, and Fernando Pereira (2005). Online Large-Margin Training of Dependency Parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*. https://www.aclweb.org/anthology/P/P05/P05-1012.pdf.

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*. https://arxiv.org/abs/1701.03980.

Emily Pitler (2014). A Crossing-Sensitive Third-Order Factorization for Dependency Parsing. *Transactions of the Association for Computational Linguistics* 2, pp. 41–54. https://transacl.org/ojs/index.php/tacl/article/view/193.

Emily Pitler, Sampath Kannan, and Mitchell Marcus (2013). Finding Optimal 1-Endpoint-Crossing Trees. *Transactions of the Association for Computational Linguistics* 1, pp. 13–24. https://www.aclweb.org/anthology/Q/Q13/Q13-1002.pdf.

Nathan Ratliff, J. Andrew (Drew) Bagnell, and Martin Zinkevich (2007). (Online) Subgradient Methods for

Structured Prediction. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AIStats)*.

Helmut Schmid (2006). Trace Prediction and Recovery with Unlexicalized PCFGs and Slash Features. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL)*. https://www.aclweb.org/anthology/P/P06/P06-1023.pdf.

Roy Schwartz, Omri Abend, and Ari Rappoport (2012). Learnability-Based Syntactic Annotation Design. In *Proceedings of COLING 2012*. http://www.aclweb.org/anthology/C/C12/C12-1147.pdf.

Libin Shen, Lucas Champollion, and Aravind K. Joshi (2007). LTAG-spinal and the Treebank. *Language Resources and Evaluation* 42.1, pp. 1–19.

Mark Steedman (2000). *The Syntactic Process*. MIT Press.

Daniel H. Younger (1967). Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control* 10.2, pp. 189–208.