Efficient Implementation of Beam-Search Incremental Parsers[*]

Yoav Goldberg
Dept. of Computer Science
Bar-Ilan University
Ramat Gan, Tel Aviv, 5290002 Israel
`yoav.goldberg@gmail.com`

Kai Zhao **Liang Huang**
Graduate Center and Queens College
City University of New York
{`kzhao@gc`, `lhuang@cs.qc`}`.cuny.edu`
{`kzhao.hf`, `liang.huang.sh`}`.gmail.com`

Abstract

Beam search incremental parsers are accurate, but not as fast as they could be. We demonstrate that, contrary to popular belief, most current implementations of beam parsers in fact run in $O(n^2)$, rather than linear time, because each state-transition is actually implemented as an $O(n)$ operation. We present an improved implementation, based on *Tree Structured Stack* (TSS), in which a transition is performed in $O(1)$, resulting in a real linear-time algorithm, which is verified empirically. We further improve parsing speed by sharing feature-extraction and dot-product across beam items. Practically, our methods combined offer a speedup of $\sim$2x over strong baselines on Penn Treebank sentences, and are orders of magnitude faster on much longer sentences.

1 Introduction

Beam search incremental parsers (Roark, 2001; Collins and Roark, 2004; Zhang and Clark, 2008; Huang et al., 2009; Huang and Sagae, 2010; Zhang and Nivre, 2011; Zhang and Clark, 2011) provide very competitive parsing accuracies for various grammar formalisms (CFG, CCG, and dependency grammars). In terms of purning strategies, they can be broadly divided into two categories: the first group (Roark, 2001; Collins and Roark, 2004) uses soft (aka probabilistic) beams borrowed from bottom-up parsers (Charniak, 2000; Collins, 1999) which has no control of complexity, while the second group (the rest and many more recent ones) employs hard beams borrowed from machine translation (Koehn, 2004) which guarantee (as they claim) a linear runtime $O(kn)$ where k is the beam width. However, we will demonstrate below that, contrary to popular belief, in most standard implementations their actual runtime is in fact $O(kn^2)$ rather than linear. Although this argument in general also applies to dynamic programming (DP) parsers,[1] in this paper we only focus on the standard, non-dynamic programming approach since it is arguably still the dominant practice (e.g. it is easier with the popular arc-eager parser with a rich feature set (Kuhlmann et al., 2011; Zhang and Nivre, 2011)) and it benefits more from our improved algorithms.

The dependence on the beam-size k is because one needs to do k-times the number of basic operations (feature-extractions, dot-products, and state-transitions) relative to a greedy parser (Nivre and Scholz, 2004; Goldberg and Elhadad, 2010). Note that in a beam setting, the same state can expand to several new states in the next step, which is usually achieved by *copying* the state prior to making a transition, whereas greedy search only stores one state which is modified *in-place*.

Copying amounts to a large fraction of the slowdown of beam-based with respect to greedy parsers. Copying is expensive, because the state keeps track of (a) a stack and (b) the set of dependency-arcs added so far. Both the arc-set and the stack can grow to $O(n)$ size in the worst-case, making the state-copy (and hence state-transition) an $O(n)$ operation. Thus, beam search implementations that copy the entire state are in fact quadratic $O(kn^2)$ and not linear, with a slowdown factor of $O(kn)$ with respect to greedy parsers, which is confirmed empirically in Figure 4.

We present a way of decreasing the $O(n)$ transition cost to $O(1)$ achieving strictly linear-time parsing, using a data structure of **Tree-Structured Stack** (TSS) that is inspired by but simpler than the graph-structured stack (GSS) of Tomita (1985) used in dynamic programming (Huang and Sagae, 2010).[2] On average Treebank sentences, the TSS

[*]Supported in part by DARPA FA8750-13-2-0041 (DEFT).

[1]The Huang-Sagae DP parser (`http://acl.cs.qc.edu`) does run in $O(kn)$, which inspired this paper when we experimented with simulating non-DP beam search using GSS.

[2]Our notion of TSS is crucially different from the data

input:	$w_0 \dots w_{n-1}$
axiom	$0 : \langle 0,\ \epsilon \rangle : \emptyset$
SHIFT	$\dfrac{\ell : \langle j,\ S \rangle : A}{\ell + 1 : \langle j+1,\ S \vert w_j \rangle : A}\ j < n$
REDUCEL	$\dfrac{\ell : \langle j,\ S \vert s_1 \vert s_0 \rangle : A}{\ell + 1 : \langle j,\ S \vert s_0 \rangle : A \cup \{ s_1 \curvearrowleft s_0 \}}$
REDUCER	$\dfrac{\ell : \langle j,\ S \vert s_1 \vert s_0 \rangle : A}{\ell + 1 : \langle j,\ S \vert s_1 \rangle : A \cup \{ s_1 \curvearrowright s_0 \}}$
goal	$2n - 1 : \langle n,\ s_0 \rangle : A$

Figure 1: An abstraction of the arc-standard deductive system Nivre (2008). The stack S is a list of heads, j is the index of the token at the front of the buffer, and ℓ is the step number (beam index). A is the **arc-set** of dependency arcs accumulated so far, which we will get rid of in Section 4.1.

version, being linear time, leads to a speedup of 2x$\sim$2.7x over the naive implementation, and about 1.3x$\sim$1.7x over the optimized baseline presented in Section 5.

Having achieved efficient state-transitions, we turn to feature extraction and dot products (Section 6). We present a simple scheme of sharing repeated scoring operations across different beam items, resulting in an additional 7 to 25% speed increase. On Treebank sentences, the methods combined lead to a speedup of $\sim$2x over strong baselines ($\sim$10x over naive ones), and on longer sentences they are orders of magnitude faster.

2 Beam Search Incremental Parsing

We assume familiarity with transition-based dependency parsing. The unfamiliar reader is referred to Nivre (2008). We briefly describe a standard shift-reduce dependency parser (which is called "arc-standard" by Nivre) to establish notation. Parser *states* (sometimes called configurations) are composed of a stack, a buffer, and an arc-set. Parsing *transitions* are applied to states, and result in new states. The arc-standard system has three kinds of transitions: SHIFT, REDUCEL, and REDUCER, which are summarized in the deductive system in Figure 1. The SHIFT transition removes the first word from the buffer and pushes it to the stack, and the REDUCEL and REDUCER actions each add a dependency relation between the two words on the top of the stack (which is achieved by adding the arc $s_1 \curvearrowleft s_0$ or $s_1 \curvearrowright s_0$ to the arc-set A), and pops the new dependent from the stack. When reaching the goal state the parser returns a tree composed of the arcs in the arc-set.

At parsing time, transitions are chosen based on a trained scoring model which looks at features of the state. In a *beam* parser, k items (hypotheses) are maintained. Items are composed of a state and a score. At step i, each of the k items is extended by applying all possible transitions to the given state, resulting in $k \times a$ items, a being the number of possible transitions. Of these, the top scoring k items are kept and used in step $i+1$. Finally, the tree associated with the highest-scoring item is returned.

3 The Common Implementation of State

The *stack* is usually represented as a list or an array of token indices, and the *arc-set* as an array `heads` of length n mapping the word at position m to the index of its parent. In order to allow for fast feature extraction, additional arrays are used to map each token to its left-most and right-most modifier, which are used in most incremental parsers, e.g. (Huang and Sagae, 2010; Zhang and Nivre, 2011). The buffer is usually implemented as a pointer to a *shared* sentence object, and an index j to the current front of the buffer. Finally, it is common to keep an additional array holding the transition sequence leading to the current state, which can be represented compactly as a pointer to the previous state and the current action. The state structure is summarized below:

```
class state
   stack[n] of token_ids
   array[n] heads
   array[n] leftmost_modifiers
   array[n] rightmost_modifiers
   int j
   int last_action
   state previous
```

In a greedy parser, state transition is performed in-place. However, in a beam parser the states cannot be modified in place, and a state transition operation needs to result in a new, independent state object. The common practice is to copy the current state, and then update the needed fields in the copy. Copying a stack and arrays of size n is an

structure with the same name in an earlier work of Tomita (1985). In fact, Tomita's TSS merges the top portion of the stacks (more like GSS) while ours merges the bottom portion. We thank Yue Zhang for informing us that TSS was already implemented for the CCG parser in zpar (`http://sourceforge.net/projects/zpar/`) though it was not mentioned in his paper (Zhang and Clark, 2011).

$O(n)$ operation. In what follows, we present a way to perform transitions in $O(1)$.

4 Efficient State Transitions

4.1 Distributed Representation of Trees

The state needs to keep track of the set of arcs added to the tree so far for two reasons:

(a) In order to return the complete tree at the end.
(b) In order to compute features when parsing.

Observe that we do not in fact need to store any arc in order to achieve (a) – we could reconstruct the entire set by backtracking once we reach the final configuration. Hence, the arc-set in Figure 1 is only needed for computing features. Instead of storing the entire arc-set, we could keep only the information needed for feature computation. In the feature set we use (Huang and Sagae, 2010), we need access to (1) items on the buffer, (2) the 3 top-most elements of the stack, and (3) the current left-most and right-most modifiers of the two topmost stack elements. The left-most and right-most modifiers are already kept in the state representation, but store more information than needed: we only need to keep track of the modifiers of current stack items. Once a token is removed from the stack it will never return, and we will not need access to its modifiers again. We can therefore remove the left/rightmost modifier arrays, and instead have the stack store triplets `(token, leftmost_mod, rightmost_mod)`. The `heads` array is no longer needed. Our new state representation becomes:

```
class state
   stack[n] of (tok, left, right)
   int j
   int last_action
   state previous
```

4.2 Tree Structured Stack: TSS

We now turn to handle the stack. Notice that the buffer, which is also of size $O(n)$, is represented as a pointer to an immutable shared object, and is therefore very efficient to copy. We would like to treat the stack in a similar fashion.

An *immutable stack* can be implemented functionally as a *cons list*, where the head is the top of the stack and the tail is the rest of the stack. Pushing an item to the stack amounts to adding a new head link to the list and returning it. Popping an item from the stack amounts to returning the tail of the list. Notice that, crucially, a pop operation does not change the underlying list at all, and a push operation only adds to the front of a list. Thus, the stack operations are non-destructive, in the sense that once you hold a reference to a stack, the view of the stack through this reference does not change regardless of future operations that are applied to the stack. Moreover, push and pop operations are very efficient. This stack implementation is an example of a persistent data structure – a data structure inspired by functional programming which keeps the old versions of itself intact when modified (Okasaki, 1999).

While each client sees the stack as a list, the underlying representation is a tree, and clients hold pointers to nodes in the tree. A push operation adds a branch to the tree and returns the new pointer, while a pop operation returns the pointer of the parent, see Figure 3 for an example. We call this representation a tree-structured stack (TSS).

Using this stack representation, we can replace the $O(n)$ stack by an integer holding the item at the top of the stack (`s0`), and a pointer to the tail of the stack (`tail`). As discussed above, in addition to the top of the stack we also keep its leftmost and rightmost modifiers `s0L` and `s0R`. The simplified state representation becomes:

```
class state
   int s0, s0L, s0R
   state tail
   int j
   int last_action
   state previous
```

State is now reduced to seven integers, and the transitions can be implemented very efficiently as we show in Figure 2. The parser state is transformed into a compact object, and state transitions are $O(1)$ operations involving only a few pointer lookups and integer assignments.

4.3 TSS vs. GSS; Space Complexity

TSS is inspired by the graph-structured stack (GSS) used in the dynamic-programming parser of Huang and Sagae (2010), but without reentrancy (see also Footnote 2). More importantly, the state signature in TSS is much slimmer than that in GSS. Using the notation of Huang and Sagae, instead of maintaining the full DP signature of

$$\widetilde{\mathbf{f}}_{\mathrm{DP}}(j, S) = (j, \mathbf{f}_d(s_d), \ldots, \mathbf{f}_0(s_0))$$

where s_d denotes the d^{th} tree on stack, in non-DP TSS we only need to store the features $\mathbf{f}_0(s_0)$ for the final tree on the stack,

$$\widetilde{\mathbf{f}}_{\mathrm{noDP}}(j, S) = (j, \mathbf{f}_0(s_0)),$$

```
def Shift(state)
    newstate.s0 = state.j
    newstate.s0L = None
    newstate.s0R = None
    newstate.tail = state
    newstate.j = state.j + 1
    return newstate
```

```
def ReduceL(state)
    newstate.s0 = state.s0
    newstate.s0L = state.tail.s0
    newstate.s0R = state.s0R
    newstate.tail = state.tail.tail
    newstate.j = j
    return newstate
```

```
def ReduceR(state)
    newstate.s0 = state.tail.s0
    newstate.s0L = state.tail.s0L
    newstate.s0R = state.s0
    newstate.tail = state.tail.tail
    newstate.j = j
    return newstate
```

Figure 2: State transitions implementation in the TSS representation (see Fig. 3 for the `tail` pointers). The two lines on `s0L` and `s0R` are specific to feature set design, and can be expanded for richer feature sets. To conserve space, we do not show the obvious assignments to `last_action` and `previous`.

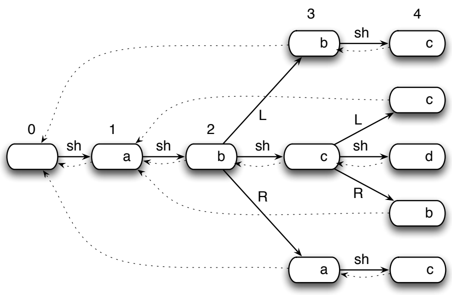

Figure 3: Example of tree-structured stack. The forward arrows denote state transitions, and the dotted backward arrows are the `tail` pointers to the stack tail. The boxes denote the top-of-stack at each state. Notice that for $b = \textit{shift}(a)$ we perform a single *push* operation getting $b.tail = a$, while for $b = \textit{reduce}(a)$ transition we perform two pops and a push, resulting in $b.tail = a.tail.tail$.

thanks to the uniqueness of `tail` pointers ("left-pointers" in Huang and Sagae).

In terms of space complexity, each state is reduced from $O(n)$ in size to $O(d)$ with GSS and to $O(1)$ with TSS,[3] making it possible to store the entire beam in $O(kn)$ space. Moreover, the constant state-size makes memory management easier and reduces fragmentation, by making it possible to pre-allocate the entire beam upfront. We did not explore its empirical implications in this work, as our implementation language, Python, does not support low-level memory management.

4.4 Generality of the Approach

We presented a concrete implementation for the arc-standard system with a relatively simple (yet state-of-the-art) feature set. As in Kuhlmann et al. (2011), our approach is also applicable to other transitions systems and richer feature-sets with some additional book-keeping. A well-documented Python implementation for the *labeled arc-eager* system with the rich feature set of Zhang and Nivre (2011) is available on the first author's homepage.

5 Fewer Transitions: Lazy Expansion

Another way of decreasing state-transition costs is making less transitions to begin with: instead of performing all possible transitions from each beam item and then keeping only k of the resulting states, we could perform only transitions that are sure to end up on the next step in the beam. This is done by first computing transition scores from each beam item, then keeping the top k highest scoring $(state, action)$ pairs, performing only those k transitions. This technique is especially important when the number of possible transitions is large, such as in labeled parsing. The technique, though never mentioned in the literature, was employed in some implementations (e.g., Yue Zhang's `zpar`). We mention it here for completeness since it's not well-known yet.

6 (Partial) Feature Sharing

After making the state-transition efficient, we turn to deal with the other major expensive operation: feature-extractions and dot-products. While we can't speed up the process, we observe that some computations are repeated in different parts of the beam, and propose to share these computations. Notice that relatively few token indices from a state can determine the values of many features. For example, knowing the buffer index j determines the words and tags of items after location j on the buffer, as well as features composed of combinations of these values.

Based on this observation we propose the notion of a *state signature*, which is a set of token indices. An example of a state signature would be $\text{sig}(state) = (\texttt{s0}, \texttt{s0L}, \texttt{s1}, \texttt{s1L})$, indicating the indices of the two tokens at the top of the stack together with their leftmost modifiers. Given a sig-

[3]For example, a GSS state in Huang and Sagae's experiments also stores `s1`, `s1L`, `s1R`, `s2` besides the $\mathbf{f}_0(s_0)$ features (`s0`, `s0L`, `s0R`) needed by TSS. d is treated as a constant by Huang and Sagae but actually it could be a variable.

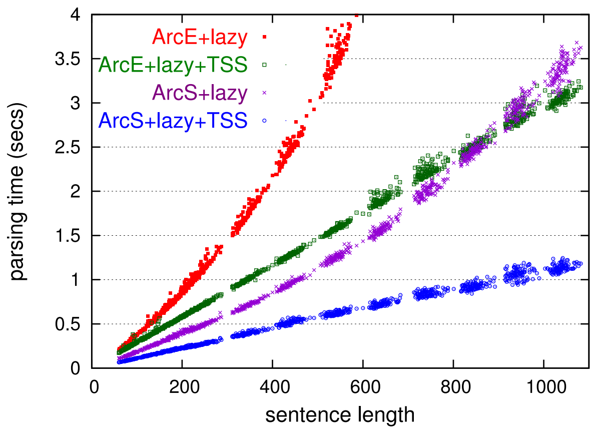

Figure 4: Non-linearity of the standard beam search compared to the linearity of our TSS beam search for labeled arc-eager and unlabeled arc-standard parsers on long sentences (running times vs. sentence length). All parsers use beam size 8.

nature, we decompose the feature function $\phi(x)$ into two parts $\phi(x) = \phi_s(\text{sig}(x)) + \phi_o(x)$, where $\phi_s(\text{sig}(x))$ extracts all features that depend exclusively on signature items, and $\phi_o(x)$ extracts all other features.[4] The scoring function $\mathbf{w} \cdot \phi(x)$ decomposes into $\mathbf{w} \cdot \phi_s(\text{sig}(x)) + \mathbf{w} \cdot \phi_o(x)$. During beam decoding, we maintain a cache mapping seen signatures $\text{sig}(state)$ to (partial) transition scores $\mathbf{w} \cdot \phi_s(\text{sig}(state))$. We now need to calculate $\mathbf{w} \cdot \phi_o(x)$ for each beam item, but $\mathbf{w} \cdot \phi_s(\text{sig}(x))$ only for one of the items sharing the signature. Defining the signature involves a natural balance between signatures that repeat often and signatures that cover many features. In the experiments in this paper, we chose the signature function for the arc-standard parser to contain all core elements participating in feature extraction[5], and for the arc-eager parser a signature containing only a partial subset.[6]

7 Experiments

We implemented beam-based parsers using the traditional approach as well as with our proposed extension and compared their runtime.

The first experiment highlights the non-linear behavior of the standard implementation, compared to the linear behavior of the TSS method.

[4]One could extend the approach further to use several signatures and further decompose the feature function. We did not pursue this idea in this work.

[5]`s0,s0L,s0R,s1,s1L,s1R,s2,j`.

[6]`s0, s0L, s0R,s0h,b0L,j`, where `s0h` is the parent of `s0`, and `b0L` is the leftmost modifier of `j`.

system	plain (sec 3)	plain +TSS (sec 4)	plain +lazy (sec 5)	plain +TSS +lazy	+TSS+lazy +feat-share (sec 6)
`ArcS-U`	20.8	38.6	24.3	41.1	47.4
`ArcE-U`	25.4	48.3	38.2	58.2	72.3
`ArcE-L`	1.8	4.9	11.1	14.5	17.3

Table 1: Parsing speeds for the different techniques measured in sentences/sec (beam size 8). All parsers are implemented in Python, with dot-products in C. `ArcS`/`ArcE` denotes arc-standard vs. arc-eager, `L`/`U` labeled (stanford deps, 49 labels) vs. unlabeled parsing. `ArcS` use feature set of Huang and Sagae (2010) (50 templates), and `ArcE` that of Zhang and Nivre (2011) (72 templates).

As parsing time is dominated by score computation, the effect is too small to be measured on natural language sentences, but it is noticeable for longer sentences. Figure 4 plots the runtime for synthetic examples with lengths ranging from 50 to 1000 tokens, which are generated by concatenating sentences from Sections 22–24 of Penn Treebank (PTB), and demonstrates the non-linear behavior (dataset included). We argue parsing longer sentences is by itself an interesting and potentially important problem (e.g. for other languages such as Arabic and Chinese where word or sentence boundaries are vague, and for parsing beyond sentence-level, e.g. discourse parsing or parsing with inter-sentence dependencies).

Our next set of experiments compares the actual speedup observed on English sentences. Table 1 shows the speed of the parsers (sentences/second) with the various proposed optimization techniques. We first train our parsers on Sections 02–21 of PTB, using Section 22 as the test set. The accuracies of all our parsers are at the state-of-the-art level. The final speedups are up to 10x against naive baselines and $\sim$2x against the lazy-transitions baselines.

8 Conclusions

We demonstrated in both theory and experiments that the standard implementation of beam search parsers run in $O(n^2)$ time, and have presented improved algorithms which run in $O(n)$ time. Combined with other techniques, our method offers significant speedups ($\sim$2x) over strong baselines, or 10x over naive ones, and is orders of magnitude faster on much longer sentences. We have demonstrated that our approach is general and we believe it will benefit many other incremental parsers.

References

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of NAACL*.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of ACL*.

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *Proceedings of HLT-NAACL*, pages 742–750.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of ACL 2010*.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *Proceedings of EMNLP*.

Philipp Koehn. 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Proceedings of AMTA*, pages 115–124.

Marco Kuhlmann, Carlos Gmez-Rodrguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of ACL*.

Joakim Nivre and Mario Scholz. 2004. Deterministic dependency parsing of english text. In *Proceedings of COLING*, Geneva.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

Brian Roark. 2001. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276.

Masaru Tomita. 1985. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2*, pages 756–764.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of EMNLP*.

Yue Zhang and Stephen Clark. 2011. Shift-reduce ccg parsing. In *Proceedings of ACL*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of ACL*, pages 188–193.