Treebank Construction by Levels Using Constrained Chart Parsing

Seth Kulick
Linguistic Data Consortium
Institute for Research in Cognitive Science
University of Pennsylvania
skulick@cis.upenn.edu

Dan Bikel
IBM Thomas J. Watson
Research Center
Yorktown Heights, NY
dbikel@us.ibm.com

Anthony Kroch
Department of Linguistics
University of Pennsylvania
kroch@change.ling.upenn.edu[*]

October 30, 2006

Abstract

We describe a method of treebank annotation that partitions the annotation process into several levels of treebanking. This requires the integration of hard constraints respecting earlier levels of bracketing into the parser. We discuss the implementation of such prebracketing in a CKY-style parser, the Bikel parsing engine. Experiments on the Penn Treebank and the Penn-Helsinki Parsed Corpus of Early Modern English demonstrate the importance of run-time beam widening, without which constraint-satisfying parses could not be constructed.

1 Introduction

Treebank construction typically uses a parser as first step, in which a sentence is parsed automatically and then corrected by a treebanker. In this paper we discuss an alternative process, in which there are different levels of treebank annotation,

[*]We would like to thank Ann Bies, Ryan Gabbard, Mitch Marcus, Mark Liberman, Beatrice Santorini, and two anoymous reviewers for helpful comments and discussion. This work was funded by National Science Foundation grant BCS-0527116.

Hajič, J. and Nivre, J. (eds.): Proceedings of the TLT 2006, pp. 43–54.

so that the first level is focused only on some subset of the full structure, such as only S brackets. Then parsing can proceed, treating those S brackets as “gold”, and only returning parses that are consistent with those brackets. Another treebanker can then correct just the NP brackets (for example), and the parsing proceeds again, now respecting both the S and NP brackets, and so on.

While this now requires several levels of treebanking instead of just one, it has the potential to be more efficient overall if the parses for the final stage are significantly more accurate than those done without some gold nonterminals prebracketed, so that the amount of work required for the final detailed treebanking is much less. This becomes even more significant when using code for recovering empty categories, as a way to further lessen the work for the final stage of treebanking. Research in this area has found that there is a significant loss in accuracy on parser output as compared to gold trees [10]. An extra level of simplified treebanking that leads to large increases in accuracy for full parsing could make the difference between disastrous and reasonable empty category recovery.

An additional motivating factor for this approach is that the annotators for the first levels of partial annotation do not need to be as highly trained as those who finish the annotation of the entire tree. The work can therefore be partitioned, reserving the expertise of the experienced, and more expensive, annotators only for when it is required.

In this paper we discuss development and implementation issues for this approach in the context of the creation of phrase-structure treebanks using a lexicalized chart parser. The combination of the parser and our top-down approach to constraints leads to various issues of implementation that had not yet been considered, related to the parser’s use of pruning. We illustrate these issues with experiments with two different treebanks, the Penn Treebank (PTB) and the Penn-Helsinki Parsed Corpus of Early Modern English (PPCEME). The latter is an important test case since it is the training material for the current round of historical treebank construction. Also, the parser does not function well on this material, which highlights the effect on accuracy of prebracketing certain constituents.

2 Previous Work

The seminal work with constraints in the context of a stochastic parsing model was that of [12]. The goal in that work was to treat parsing constraints as the “observed” data when performing EM, in order to train a full stochastic parsing model in the face of partially-bracketed data. This work inspired other work with parsing and EM, notably [9], which in turn inspired the EM experiments of [6].

There is also a degree of similarity between the present work and that of [5], in which an unlexicalized PCFG is used to create a derivation forest that serves as a set of constraints for a more accurate and detailed lexicalized parsing model.

The work that is probably the most immediate source of comparison is the *Annotate* tool that was developed for constructing the NEGRA corpus [4]. This tool allows the annotator to partially bracket the sentence, and interactively use the results of a parser that respects that bracketing. This work and ours share the goal of avoiding "the immediate generation of complete structures and subsequent error correction by a human" [4], although there are major differences in approaches.

Annotate is based on a bottom-up approach to tree construction and partial bracketing. At each step, the parser considers all possible new bracketings, and selects the most likely one as the next step for the annotator to accept or reject. The parser considers only those new bracketings that are consistent with what has been bracketed so far.

In contrast, our approach is more top-down and batch-based. We explicitly do not want a single treebanker to do be doing the complete annotation of a tree, even with interactive help from the parser. As mentioned in the introduction, we wish to partition the work among different treebankers, doing prebracketing of different levels of complexity.

This difference leads to a crucial implementation issue. The *Annotate* system can afford to consider all possible bracketings at each step, since it gets immediate feedback as to what the next step of the parse construction should be. In chart parsing as in [3, 7], however, the parser needs to implement a beam search and prune away potential constituents when filling each chart cell, due to the huge growth in the number of such constituents. This remains true even for chart parsing with hard constraints as in our work, leading to the concern that the parser will mistakenly prune away the constituents that would allow it to construct a parse respecting the hard constraints. This issue is discussed in detail in Section 5.

3 Constraint Specification

The constraints are specified as a tree, in which the relation between a parent and its children is understood to be non-immediate, instead of immediate, dominance. The resulting parse is required to have constituents that match those in the prebracketing, although it is otherwise free to create other constituents.

An example prebracketing is shown in Figure 1(A). This requires that there be an NP at span (0,0), an NP at span (2,4), and a S covering the whole sentence. Both Figure 1(B) and 1(C) are consistent with this prebracketing.

Since the parser input is in the form of a tree including the actual lexical items, the part of speech tags must be included as well. However, we take the constraints

```
(A)
   (S (NP (NNP John))
      (VBD ate)
      (NP (DT the) (NN ice)
          (NN cream)))
(B)
   (S (NP (NNP John))
      (VP (VBD ate)
          (NP (DT the) (NN ice)
              (NN cream))))
(C)
   (S (NP (NNP John))
      (VP (VBD ate)
          (NP (NP (DT the)
                  (NN ice))
              (NP (NN cream)))))
(D)
   (* (NP (NNP John))
      (VBD ate)
      (NP (DT the) (NN ice)
          (NN cream)))
```

Figure 1: Prebracketing example (A,D) and two possible parses (B,C)

on the bracket labels to apply only to the nonterminals, and not the POS tags. That is, while we have shown 1(B) and (C) as having the same tags as in 1(A), that does not have to be the case. The reason for this is that we do not always wish to enforce that the given tags be used, and if we do there is already a mechanism in the parser for enforcing that restriction.

While (B) and (C) are both consistent with (A), (C) has extra NPs that are not present in (A). If the annotator for a simplified level of treebanker has annotated the NPs as gold, we do not wish to allow the parser to produce additional NPs. We therefore allow the specification of *strict constraints* for certain labels, to enforce that the parses have a 100% precision for brackets with those labels, as well as recall. For example, if the strict constraint set is { NP} and we have the prebracketing in Fig. 1(A), then the only NPs allowed are those in 1(A). This rules out 1(C), while (B) is still consistent with the prebracketing *and* the strict constraint set.

In general, all the work described here will assume that the strict constraints for a prebracketing correspond to the nonterminals that are being prebracketed. For example, if NP and S are being prebracketed, then NP and S are the strict constraints. This is a separate specification, though, and does not have to be the case. One can imagine cases in which we may not wish to utilize the strict constraints. For example, it could be that treebankers for a simplified level should annotate only some of the NPs (perhaps maximal NPs), and allow the parser to hypothesize other NPs. This is a matter of experimentation and what is cognitively easiest for the treebanker, an issue currently being explored.

We allow a wildcard constraint, to specify that some bracketing exists, regardless of the label. While this can be used anywhere in the input tree, in the work that follows we only use it at the root, in order to keep the input as a tree, in case we are not prebracketing the nonterminal at the root. For example, if the treebanker is annotating only the NPs, then (A) in Figure 1 would be input to the parser as (D). In conjunction with the strict constraints, the wildcard is interpreted as “anything but one of the strict constraints”. For example, if NP is a strict constraint in (D), then the root of the parse can be anything but an NP.

4 Parsing with Constraints

In this section we give a brief account of how our constraint system interacts with the parser’s tree construction.

We first give a brief recap of the Collins parser, as implemented in the the Bikel parsing engine [3].[1] The PCFG that underlies it has rules of the form

$$P \rightarrow L_n L_{n-1} \ldots L_1 H R_1 \ldots R_{n-1} R_n \tag{1}$$

where P is the parent, H is P's head child, and L_i and R_i are the left and right modifiers. Every nonterminal is lexicalized, meaning that the label is augmented with head information $(\mathtt{w}, \mathtt{t}) = (\mathtt{head\ word}, \mathtt{head\ word\ POS\ tag})$. In (1) P, H, L_i, and R_i are all lexicalized nonterminals, and P and H have same head information. The training process decomposes trees into small steps based on (1), creating a model defined in a top-down manner, although decoding actually proceeds bottom-up. This generative model first generates P, then its head-child H, and then each of the left and right modifiers. This is a recursive process, and each newly-generated modifier is likewise treated as a parent.

4.1 Interaction of the Parser and Constraint System

With the addition of a constraint system,[2] as each new potential chart item is created, it is checked for consistency with the partial tree bracketing set of constraints. While such constraints on parsing are for the most part straightforward to implement, there are some subtleties involved, especially with the use of strict constraints, as discussed in Section 3.

The probabilistic CKY–variant for head-driven parsing has three main operations: unary projection, joining, and completion. We discuss in the following subsections how each operation interacts with the constraint system.

4.1.1 Item Projection

Unary projection allows a completed chart item to be projected up to a new nonterminal.[3] An item can be killed at this point without entering the chart if it conflicts with the constraints. For example, suppose that IP and NP are being prebracketed, and both are strict constraints. A given prebracketing is `(IP-0-10 NP-4-5 NP-8-10)`.[4] Suppose there is a chart item NP(4,5), which therefore matches the constraint `NP-4-5`, and that it projects to a new NP. That newly projected NP is

[1] Publicly available at `http://www.cis.upenn.edu/ ~dbikel/software.html`.

[2] A constraint system was already present in the Bikel parser, which allowed a complete tree to be given as parser input, which was used for the constraint experiments in [1]. The work described here takes advantage of that previous work, which already had the infrastructure for associating constraints with items, although the implementation is significantly different.

[3] A completed item is one with STOP probabilities added, meaning that it can receive no more modifiers.

[4] We use `XP-A-B` as shorthand for a constraint with bracket XP and span (A,B).

immediately eliminated as a possibility, because there is no additional NP in the constraints. Without the use of strict constraints it would be entered into the chart.

4.1.2 Joining of Items

When a modifier joins to a modificand, the new item is checked for two possible ways of violating the partial tree constraints. One case is when the new item intersects with a constraint. For example, suppose that IPs are being prebracketed, and the prebracketing is[5]

```
(IP (CONJ and) (VBD left)
    (D this)(N letter)
    (IP (TO to) (BE be)
        (VAN sent) ...))
```

and the modificand is `(IP (TO to))`, and the modifier (to its left) is `(NP (N letter))`. A constituent containing `... (N letter) (TO to)...` is incompatible with the specified `(IP (TO to)...` constraint and so this constituent is killed.

The second possibility is somewhat more subtle. The new item is killed if some constraint that should have been satisfied within that item is not satisfied. For example, suppose that the prebracketing contains

```
(VP-3-11 VBD-3-3 (NP-4-11 ...))
```

If a modificand `(VP (VBD))` with span (3,3) joins with a modifier `(NP ...)` with span (4,11) to form a VP with span (3,11), that is okay since the constraint `NP-4-11` is satisfied within the new item. But it might be that the same modificand joins first with an NP with span (4,9) and then with a NP with span (10,11) to form a VP with span (3,11). Upon the last joining, constraints within (3,11) will be checked and this item will be rejected since `NP-4-11` has no match. Note that such items must be killed before entering the chart since an item that does not satisfy some constraint could have a higher probability than an “equivalent” item that satisfies the constraints, and so checking at the time of item completion is not sufficient.[6]

4.1.3 Item completion

As mentioned above, items are completed when they have STOP probabilities added, thus finishing modifier attachment. At this point “spurious” projections,

[5] This example and some of the following are taken from the PPCEME, which is why the tags and nonterminals are somewhat different than in the Penn Treebank.

[6] This could be made somewhat more efficient. For example, in this case we know as soon as the NP with span (4,9) attaches as a modifier to the VP at (3,3), that the item is hopeless.

that violate the strict interpretation of the constraints, can be killed. For example, if the partial bracketing specifies only a `NP-3-6`, and item completion has made a new NP at span (4,6), this is killed. Before it had been completed, it could have "grown" into a constituent with span (3,6) with left modifier attachment, but at the point of completion it violates the strict constraints.

4.2 Internal Transformations and Prebracketed Constraints

One notable aspect of the Collins parser is that it applies various transformations, detailed in [2], to the training data, and the chart items constructed during a parse of course correspond to this model. The usual procedure is that at the end of a parse, various operations undo the effect of such transformations to be more in accord with what the trees are expected to be. A simple case is the SG transformation, in which S nodes without a subject are changed to SG in the training data. The parser can therefore create chart items with label SG. This creates a slight issue with specifying hard constraints on constituents, since human annotators, certainly, would not wish to worry about what S constituent is internally an S or a SG for the parser. However, this is a trivial matter to handle, since we can just assume that S and SG are equal for purposes of comparison.

A more serious case is that of the base (nonrecursive) NPs. The parser makes special treatment of base NPs, for various reasons that are not of concern here. (For details, see [2, 8]) What does matter is that at the end of a parse, a NPB (base NP) that is a unary child of an NP is removed.[7] When a NPB is not a unary child of an NP, it is renamed as NP in the final postprocessing of the parse. A NPB is almost always the child of a NP, so these are the two possibilities. This creates the unpleasant situation in which a single NP in the prebracketing could correspond to two different nodes in the parse tree as it is being constructed.

For example, suppose that the parser creates the subtree `(NP (NPB ...))` at span (2,4). At the end of the parser, the unary child NPB will be removed, resulting in `(NP ..)` at span (2,4). But now suppose that the prebracketing specifies the constraint NP-2-4. The NPB constituent satisfies this constraint, and so when it projects up to the higher NP, this NP will be killed under the strict interpretation of the constraints, as described in Section 4.1.1.[8] But this is not desired action, since at the end the NP and NPB would collapse to one NP, giving just the bracketing specified by the constraint. We handle this problem by making a straightforward and somewhat brute force change to the implementation, basically handling this as

[7] That is, the NPB node is removed, with the NPB's children becoming the children of the NP. The NPB subtree is not removed.

[8] If the constraint prebracketing specified two NPs, as in `(NP-2-4 (NP-2-4 ..))`, then this would allow the new NP projection.

a special case.[9]

One can consider, of course, this whole problem to be a nonissue, insisting that prebrackets be done in terms of NPB and NP, and in general more closely match the model the parser is actually using. In situations like [1], in which the parser is being constrained to exactly recreate a parse done before, this is entirely appropriate and possible. Our situation, in which the prebracketing may be done by treebankers who have no interest in what the parser is doing internally, makes it quite difficult to rely on prebracketing that exists in the internal, transformed tree space. There is a conflict here, and the greater the difference between the transformed trees used by the model and their detransformed counterparts, the greater the conflict is. We view the fixups done to handle the base NP case as the outer limit of what is reasonable to attempt to do.

5 Experiments

We describe here however experiments that have been run on both the Penn Treebank [13] and the PPCEME corpus. The purposes were several:

- To verify that the prebracketing code is working as expected.
- To better understand the effects of forcing certain constituents to be gold.
- To determine how many sentences fail to parse with the reduced search space determined by the prebracketing.

The last point was the greatest concern. If, for example, a treebanker annotates the S brackets, but the parser is unable to come up with a parse respecting those brackets for a significant number of sentences, this defeats the goal of lessening the work for the final stage of treebanking. What we have found is that while this is indeed an issue, it can be overcome by beam widening and relaxing certain of the parser's internal hard constraints (independent of the hard constraints specified by the prebracketing).

Aside from the relevance of the PPCEME corpus as the training material for current historical treebank development, it is also a good test case for comparison to the PTB, since the parser is working at a significantly lower level of accuracy on the PPCEME compared to the PTB.

[9] It actually gets even messier than described above, since if there were modifiers attached to the higher NP (sisters to the NPB), then the higher NP would *not* be deleted at the end, and it truly would violate the specified constraints. This case is also handled by our fix.

	N	WB	R	WB +R
None	0			0
S	35	2	25	2
NP	143	12	115	4
S&NP	211	36	167	27

Table 1: Nulls with PTB run for Neither(N), Wide Beam (WB), Relax (R), and Wide Beam+Relax (WB+R)

	none	S	NP	S&NP
all	88.53/88.63	92.27/92.09	95.21/94.45	97.17/96.48
NP	91.03/89.90	92.02/90.60	99.52/98.84	99.53/99.11
VP	90.51/90.35	96.57/96.42	93.77/92.56	97.22 96.43
S	89.42/89.52	100.0/99.79	92.96/90.57	100.0/99.78
PP	89.41/84.58	86.88/86.11	97.72/97.06	98.14 96.74
SBAR	84.08/87.29	94.05/97.24	89.46/89.71	94.13 95.75

Table 2: Recall/Precision for Section 23 with Wide Beam and Relaxation

5.1 Experiments with PTB

As is usual, we trained on sections 2-21, and tested on the 2416 sentences in Section 23. In addition to parsing with no prebracketing, we parsed with the gold S and NP brackets separately and also together. Parser evaluation was done with the `evalb` program, following the practice in [7] and elsewhere.

The first column of Table 1 shows the number of nulls resulting from runs for the various of prebracketing, all run without a wider beam or relaxation. As can be seen, there is a huge increase in the number of nulls, particularly with the NP bracketing and the S&NP bracketing together.

Parsing with an expanded beam, with the results shown in the second column, had a significant effect on reducing the number of nulls.[10] We also tried a "parser relaxation", in which if no parse is found even at the maximum beam setting, zero probability estimates are instead given a very low probability. This has some effect, although not as much as the beam widening. The best results were obtained with both beam widening and the relaxation. The most severe case, that of S and NP prebracketing, falls from 211(8.7%) to 27 (1.1%) nulls. We take the importance of the wide beam to be a manifestation of how often the correct parse is lost to pruning in regular parsing.

Table 2 shows the recall/precision results for the various levels of prebracketing, for the runs with the wide beam and relaxation. The rows break down the

[10] The prune factor was increased from 9.2 to 69.1, or 4 to 30 in terms of the maxPruneFactor setting in the Bikel parser. The parser first tries parsing with a prune factor of 4, and then if that fails it tries 4.5, etc. It is very likely that the beam does not need to be expanded this much to obtain the reduction in nulls, but this has not been experimented with yet.

	N	WB+R
None	46	17
IP		26
NP		32
IP&NP		74

Table 3: Nulls with PPCEME run for Neither(N), and Wide Beam+Relax (WB+R)

	none	IP	NP	IP&NP
all	77.22/ 76.93	88.36/ 86.64	89.73/ 88.48	96.14/ 93.48
NP	83.50/ 81.75	86.51/ 83.32	99.70/ 98.08	99.69/ 98.26
IP	72.23/ 72.19	100.00/ 99.97	79.94/ 78.64	100.00/ 99.97
PP	78.20/ 78.14	85.63/ 85.25	93.61/ 93.25	96.54/ 95.00
CP	61.67/ 63.42	91.73/ 90.09	75.54/ 73.56	92.83/ 85.93

Table 4: Recall/Precision for PPCEME with Wide Beam and Relaxation

results for the most frequent nonterminals.[11]

5.2 Experiments with PPCEME

The Penn-Helsinki Parsed Corpus of Early Modern English [11] is a corpus of nearly 1.8 million words of texts from three different time periods within the years 1500-1710. For the experiments described here, we used only the texts from the latest time period, 1640-1710, which consists of about 500K words. We did a balanced 80/10/10 split, and the results reported are for training on the 80% section, and testing on one of the 10% sections. There were 28603 sentences used, with each 10% section having 2860 sentences.

This corpus was created for research in historical linguistics, and as can be expected, the annotation style is somewhat different than for the PTB. We will not go into detail here on those differences, since they are not particularly relevant for the purposes of this paper, except to note that IP is used instead of S, and CP instead of SBAR.[12] Table 3, analogous to Table 1, shows the number of nulls resulting from various levels of prebracketing. Again analogous to the PTB work, we ran with either no prebracketing, IP, NP, or IP and NP together. Since there were already a significant number of nulls for no prebracketing without wide beam or relaxation, we concentrated on the runs with both relaxation and a wide beam

[11] Note that the S bracketing result for the S and S&NP prebracketing is not quite 100% precision. Likewise, the score for NP brackets with NP and S&NP prebracketing is not quite 100%. Space does not permit adequate discussion here, but there are a few special cases that the code is not handling properly, which need to be corrected. We highly doubt that the corrections will have any effect on the main points of this section, regarding the number of nulls and parsing accuracy.

[12] Also, there is no VP, which is why it doesn't appear in Table 4.

(using the same settings as for the PTB experiments), and did not experiment with using relaxation or a wide beam separately.

As can be seen, even with no prebracketing, the number of nulls is reduced from 46 to 17, an indication perhaps of just how badly tuned the parsing model currently is to this corpus.[13] The number of nulls rises with both types of prebracketing, reaching 74 (2.6%), compared with 17 (0.6%) with no prebracketing.

Table 4 shows the recall/precision results for the various levels of prebracketing, analogous to Table 2. As can be seen, there is a tremendous increase in accuracy, even just with only IP bracketing. As can be expected, correct IP bracketing leads to an increase in CP accuracy, and even in PP accuracy. The increase with just NP bracketing is also quite significant.

Our conclusion is that while the reduction in nulls is not ideal, it is low enough so that together with the huge increase in accuracy, this use of prebracketing for treebank construction is feasible, and will soon be used for treebank construction. Even just with one of the easiest types of simplified treebanking, annotating just the IPs, the number of nulls is only 26 (0.9%), while the parser accuracy increases from 77.22/76.93 to 88.36/86.64[14]

6 Conclusion and Future Work

We have described in this paper a method for integrating partial tree hard constraints into the Bikel parser. We have also shown some of the consequences of this approach for parsing, and the implications of using it for a new approach to treebank construction.

In the future, as part of using this work for treebank construction, we will be experimenting with which constituents to prebracket. This will be based primarily on what is cognitively the best for the treebankers, although the effects on the parser always have to be balanced against this. For example, it might be easier to treebank the various levels of clausal annotation, such as IP and CP, together. One major, and obvious case, which we have not yet experimented with, is prebracketing conjuncts, given the problems that coordination ambiguity presents for the parser. Since the prebracketing described here used as input all the gold IPs, etc., it included all cases of IPs as conjuncts. One could imagine however having a treebanker simply mark off all conjuncts, without regard for what the label might

[13] For the PTB, expanding the beam and relaxation made no difference at all for the run with no prebracketing, since the parser found a parse for every sentence within the smallest beam, without any relaxation.

[14] It is likely that one reason for this is that the sentences in the PPCEME tend to be quite longer than in the PTB, and prebracketing the IPs eliminates a major area of parser confusion.