The Java Version of the C&C Parser
Version 0.95

Stephen Clark
with Darren Foong, Luana Bulat and Wenduan Xu

University of Cambridge Computer Laboratory

August 14, 2015

Acknowledgements

The first Java version of the C&C parser was created by Stephen Clark, based heavily on the C++ implementation of C&C. This first attempt had a working version of the DP-based chart parser, but a buggy version of the beam-search chart parser. Darren Foong, for his 2014/15 Part II project in the Computer Laboratory [15], did an outstanding job in taking this buggy version and producing a working beam-search parser, as well as considerably cleaning up the Java code. Luana Bulat spent considerable time in producing a new markedup file covering all the lexical categories in CCGbank, one of the features of the new parser. Wenduan Xu provided invaluable assistance along the way.

Thanks to Laura Rimell, a long-suffering user of C&C and collaborator of the first author, for acting as a sounding board while the Java version was being built. Thanks to Yue Zhang, whose original shift-reduce beam search parser was the precursor to the chart-based beam search parser presented here; and Ryan McDonald, whose Cambridge NLIP seminar in 2014 provided the inspiration for appying beam search to the chart.

Thanks to Julia Hockenmaier for helping Luana create the new markedup file, and of course for creating CCGbank, without which none of this research would exist. Thanks to Mark Steedman, whose shoulders we have been standing on for some time, and finally James Curran, whose original design for C&C is heavily reflected in this new version, and from whom the first author learnt most of what he knows about practical computing.

Chapter 1

Introduction

The C&C CCG parser [8, 13] was designed with efficiency in mind, with some cost to the readability and usability of the code base. This Java version — referred to as Java C&C — is designed to remedy that shortcoming to some extent, whilst retaining the goal of having a wide-coverage parser which is efficient enough to be applied to large amounts of text. Hence, the design of this initial version has been closely based on the C&C design, with an attempt to add more comments, and in general produce a Java code base which is easier to modify than the C++ version.

This initial version has been focused on providing a platform for carrying out experiments with CCGbank, rather than providing a general-purpose parsing and tagging tool (which was one of the goals of C&C). Hence the input and output interfaces are not as advanced as those in C&C, in the following respects:

- Java C&C does not have an integrated supertagger, and requires supertagged (and POS tagged) sentences as input.
- Java C&C only produces CCG-style dependencies [8] as output, whereas C&C has a number of output representations, including RASP-style grammatical relations [3] and CCG derivations which can act as input to the semantic analysis tool Boxer [2].

Possible future modifications to Java C&C include providing some of these output modalities, in particular the interface to Boxer (which is a relatively easy modification).

Java C&C also offers some improvements over C&C (as might be expected given the 10 year gap between the two), in the following respects:

- The parser's lexicon, encoded in the so-called *markedup* file, has been extended to cover all ≈ 1,200 lexical categories in CCGbank, compared to the 500 or so in C&C. Oracle experiments in Section 3.3 show that the grammar in Java C&C is now extremely close to the grammar implicit in CCGbank,[1] and has extremely good coverage on unseen (CCGbank) data.

- In addition to the Viterbi DP-style chart-based decoder, Java C&C also has a beam search chart-based decoder, which now is the default decoder. The advantages of beam search are that a) there is no need to store the complete chart; and b) there is no restriction on the locality of the features used by the model. The disadvantage, of course, is that the search is no longer optimal, but that is offet by the extra flexibility in defining feature templates. The beam search decoder also has an additional "skimmer" which produces dependency output even when there is no spanning analysis, leading to 100% coverage.

- The accuracies reported here for parsing CCGbank are higher than those reported in Clark and Curran (2007) (CL-07) [8].

Since CCG is a highly lexicalised grammar formalism [24], most of the grammar resides in the lexicon, plus a small number of manually defined combinatory rules; however, there are some elements of the grammar in (Java) C&C which are more akin to phrase-structure rules. The full grammar in (Java) C&C consists of the following:

- the set of lexical categories, plus the annotation on those categories which defines the CCG-style predicate-argument dependencies, encoded in the markedup file;

- the manually defined combinatory rule schema;

- the type-raising rules, in (Java) C&C defined as a set of type-raising rule instances;

- the unary type-changing rules [17];

- the additional rules to deal with punctuation, binary type changing, and "special" rules to deal with the small number of rule instances from CCGbank which do not conform to the combinatory rule schema;

- the set of rule instances which are optionally used to constrain the search (resulting in a context-free grammar defined in terms of those rules).

Following the C&C perceptron model [7], all models in this report have been trained using the structured perceptron [11], in the beam search case by applying the max-violation framework of Huang et al. (2012) [18, 12]. The code for the log-linear model in CL-07 is still there in Java C&C, including the inside-outside calculations required for the dependency model, but it has not been tested.

[1]The CCGbank grammar is only implicit because CCGbank is a set of syntactic analyses, from which any grammar must be extracted.

One feature of Java C&C is that all the models are trained using the CCG dependency structures from CCGbank as the gold-standard training data, following the dependency model of CL-07. The alternative is to use the normal-form derivations in CCGbank as the training data, and induce a normal-form model which models derivations, as in Hockenmaier's generative model [16] and the normal-form model from CL-07. For a dependency model, the derivations represent hidden structure in the training data, which in CL-07 are summed over (using dynamic programming over the chart). Here we apply the "hidden perceptron" [19], which is a structured perceptron adapted to the hidden variable case.

As described in Xu et al. (2014) [26], dependency models are desirable for a number of reasons:

- modeling dependencies provides an elegant solution to the "spurious" ambiguity problem in CCG [8];
- obtaining training data for dependencies is likely to be easier than for syntactic derivations, especially for incomplete data [23];
- Clark and Curran (2006) [6] show how the dependency model from CL-07 extends naturally to the partial-training case, and also how to obtain dependency data cheaply from gold-standard lexical category sequences alone;
- it has been argued that dependencies are an ideal representation for parser evaluation, especially for CCG [4, 9], and so optimizing for dependency recovery makes sense from an evaluation perspective.

Hence, in theory the parser described here could be derived almost from the predicate-argument dependencies in CCGbank alone, without recourse to the derivations at all. In practice, however, the derivations provide the following information:

- the lexical category sequences used to train the supertagger;
- the rule instances used for type-raising (although these sets are so small they could reasonably be specified by hand);
- the rule instances used in the unary and binary type-changing cases, and the punctuation rules;
- the complete set of rule instances optionally used to constrain the search.

The rest of this report is structured as follows. Chapter 2 is a short chapter describing the role of the supertagger in the parser, for both training and testing. Chapter 3 describes the various elements of the grammar (listed above), followed by a series of oracle experiments designed to test how well the grammar in Java C&C mirrors that implicit in CCGbank (in terms of how well the grammar can recover the CCG predicate-argument dependencies), and also the coverage of the grammar on unseen (CCGbank) data. Chapter 4 gives results for the Viterbi decoder, which is essentially a hidden-variable version of the C&C perceptron model [7]. Chapter 5 describes how the max-violation perceptron can be applied to the hidden-variable perceptron model, using additional feature templates in conjunction with the beam-search decoder.

The second half of the report has a series of Appendices, one for each chapter, giving more of the low-level details regarding how to carry out the experiments and replicate the results in this report.

Chapter 2

The Supertagger

The supertagger is used to provide input to the parser, at both training and test time. To provide input to the parser training process, a "jacknifed" training process for the supertagger is required, so that the output of the supertagger is representative of that seen at test time. What this means in practice is that, when supertagging some chunk of (parser) training data, the supertagger must not have been trained on that chunk. The details of the jacknifed process, and the arguments given to the supertagger training program, are given in Appendix B.

Two key parameters for the supertagger are β and K: β determines how many supertags, on average, are assigned to each word; and K determines the minimum word frequency for which the tag dictionary applies.[1] The goal, when setting both of these parameters, is to handle the trade-off between accuracy and ambiguity levels. Here the values were set to $\beta = 0.01$ and $K = 100$. This is a departure from C&C usage, where a lower β value — leading to more categories — and a lower K value — leading to fewer categories — were typically used for training.

Results for the supertagger, on both training and development data, are given in Table 2.1, including the 10 splits for the training data. The result for wsj02-21.stagged.1, for example, uses a (single-tag) supertagger trained on splits 2-10 and tested on split 1 (with the default $K = 20$). The result for wsj02-21.stagged.0.01.100.1 uses the same model but with the multitagger, with a β value of 0.01 and a K value of 100. As can be seen from the table, these values of β and K provide a reasonable compromise between accuracy and ambiguity: multitag accuracy levels at almost 99% with fewer than 2 supertags per word, on average.

The accuracies for section 00 are slightly better than those reported in the CL-07 paper, although no significance tests have been carried out. The difference between the new and old supertaggers is that the new one uses all lexical

[1] Words whose frequency is greater than K in the training data can only be assigned supertags seen with that word in the training data [20]. The terms *supertag* and *lexical category* will be used interchangeably throughout the report.

Data	tags/word	word acc	sent acc
wsj02-21.stagged.1	1.00	92.84	35.90
wsj02-21.stagged.2	1.00	92.33	33.32
wsj02-21.stagged.3	1.00	92.65	35.09
wsj02-21.stagged.4	1.00	92.91	36.38
wsj02-21.stagged.5	1.00	92.96	37.14
wsj02-21.stagged.6	1.00	92.97	37.62
wsj02-21.stagged.7	1.00	92.58	33.96
wsj02-21.stagged.8	1.00	92.90	36.51
wsj02-21.stagged.9	1.00	92.92	35.95
wsj02-21.stagged.10	1.00	92.40	34.31
wsj02-21.stagged.all	1.00	92.74	35.62
wsj02-21.stagged.0.01.100.1	1.89	98.76	80.86
wsj02-21.stagged.0.01.100.2	1.92	98.62	80.03
wsj02-21.stagged.0.01.100.3	1.89	98.65	79.80
wsj02-21.stagged.0.01.100.4	1.88	98.71	80.59
wsj02-21.stagged.0.01.100.5	1.90	98.70	81.37
wsj02-21.stagged.0.01.100.6	1.91	98.78	81.49
wsj02-21.stagged.0.01.100.7	1.89	98.69	79.27
wsj02-21.stagged.0.01.100.8	1.89	98.81	81.82
wsj02-21.stagged.0.01.100.9	1.88	98.85	82.55
wsj02-21.stagged.0.01.100.10	1.91	98.57	78.46
wsj02-21.stagged.0.01.100.all	1.90	98.71	80.63
wsj00.stagged	1.00	92.72	37.27
wsj00.stagged.0.01.100	1.89	98.82	82.96

Table 2.1: Supertagger accuracy for the various training data splits, and on the development data; sent acc is the percentage of sentences tagged completely correctly

categories in the data, rather than only those that have occurred at least 10 times in the CCGbank training data.

Note that all of these results use gold-standard POS tags. The accuracy of the supertagger, and resulting parser, drops significantly when moving to automatically assigned POS tags [14]. Wenduan Xu has recently developed an RNN-based supertagger [25], which is significantly more accurate than the maxent-based C&C supertagger when C&C uses automatically assigned POS tags (the RNN supertagger does not use POS tags). In practice, when using the parser in an application scenario, we recommend using Wenduan's supertagger to provide input to the parser. We leave it to future work to evaluate the accuracy of the parser when using such input.

Chapter 3

The Grammar

The grammar is essentially the same as the one used in C&C, except that the lexicon (the markedup file) has been extended to cover all $\approx$ 1,200 lexical categories, extended from the 500 or so covered in C&C.

3.1 The markedup File

The markedup file in C&C only covered the 500 or so lexical categories occurring at least 10 times in sections 2-21, since the original markedup file was created by hand. For Java C&C, Luana Bulat has produced a new markedup file covering the complete set of lexical categories, using a semi-automatic process which for the local dependencies is straightforward, but for the non-local dependencies requires the co-indexing information provided in the CCGbank manual. More details of this process are given in Appendix C.

3.2 The Rule Instances

This is the set of rules used with the `seen_rules` parameter in C&C, which restricts the parser to only apply rule instances it has seen in the training data (making the grammar context free). CL-07 showed this restriction to be highly useful in practice – significantly increasing the speed of the parser without losing accuracy or coverage (at least when parsing newspaper text [22]). Setting the Java C&C `seenRules` parameter to `true` is advisable when using the chart parser to parse sentences from CCGbank. Appendix C describes how this file was obtained.

3.3 Oracle Experiments

A useful test of the grammar is to see how many dependencies could possibly be recovered from the training data, if the parser had the perfect model. One such

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 97.97% (38801 of 39604 sentences parsed)

cats:  100.00% (902813 of 902813 tokens correct)
csent: 100.00% (38801 of 38801 sentences correct)

lp:    99.68% (805542 of 808154 labelled deps precision)
lr:    99.76% (805542 of 807492 labelled deps recall)
lf:    99.72% (labelled deps f-score)
lsent: 93.09% (36121 of 38801 labelled deps sentences correct)
```

Table 3.1: Coverage and oracle F-score when the input is gold-standard supertags only; coverage is the percentage of sentences which received a fully spanning parse

test is to provide the gold-standard lexical categories to the parser, and then use an oracle decoder which returns the set of dependencies with the highest F-score relative to the gold-standard set for each sentence. Auli shows how to modify the definition of equivalence in the chart so that the usual Viterbi decoder can return a derivation giving this optimal set of dependencies [1].

Table 3.1 gives the results of this oracle test for the training data. What is being tested here is how well the grammar used by the parser matches that implicit in CCGbank. For the missing 2.03% coverage, only 0.24% is due to the maximum number of categories being exceeded (here set to 1,000,000), and 1.79% is due to failures in the grammar.[1] However, the lack of coverage is a little misleading, since, in practice, the parser is likely to build some spanning analysis when provided with more than one supertag per word. Hence, in a further oracle study, the input gold-standard supertags were augmented with supertags provided by the supertagger. The supertagger was trained in a jacknifed fashion, as described in Appendix B. The parameter settings were $\beta = 0.01$ and $K = 100$ (giving the supertagger accuracy results in Table 2.1). Table 3.2 gives the results. Note that the coverage now is 99.68%, with only a slight loss in F-score.

What these results show is that the Java C&C grammar matches CCGbank extremely well, with a coverage score over 99.5% with the augmented supertags test, and F-scores over 99.5%, including an all-sentence accuracy score over 91%. The all-sentence accuracy score is the percentage of sentences with completely correct dependencies.

One disadvantage of this oracle method is that it further fragments the chart, creating more equivalence classes (since the equivalence test now includes the number of dependencies created by each sub-derivation). Hence we have developed an alternative oracle score and decoder, which uses the original chart. In

[1]Or failures in the CCGbank analyses, depending on one's perspective, since some of the analyses are inevitably noisy given the semi-automatic process used to convert the Penn Treebank trees [17].

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 99.68% (39477 of 39604 sentences parsed)

cats:  99.90% (921804 of 922747 tokens correct)
csent: 98.37% (38832 of 39477 sentences correct)

lp:    99.55% (821726 of 825475 labelled deps precision)
lr:    99.58% (821726 of 825205 labelled deps recall)
lf:    99.56% (labelled deps f-score)
lsent: 91.83% (36251 of 39477 labelled deps sentences correct)
```

Table 3.2: Coverage and oracle F-score when the input is gold-standard supertags augmented with automatically assigned supertags

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 98.16% (38876 of 39604 sentences parsed)

cats:  100.00% (907639 of 907639 tokens correct)
csent: 100.00% (38876 of 38876 sentences correct)

lp:    99.89% (807613 of 808507 labelled deps precision)
lr:    99.48% (807613 of 811802 labelled deps recall)
lf:    99.69% (labelled deps f-score)
lsent: 93.06% (36179 of 38876 labelled deps sentences correct)
```

Table 3.3: Coverage and oracle F-score when the input is gold-standard supertags only for the high-precision oracle decoder

this decoder, the score for a sub-derivation is the number of correct dependencies, minus a large score for each incorrect dependency:

```
if (goldDeps.contains(filled) & !ignoreDeps.ignoreDependency(filled, sentence)) {
    score++;
} else if (!ignoreDeps.ignoreDependency(filled, sentence)) {
    score = score - 100;
}
```

So this decoder is essentially a high-precision decoder. Table 3.3 gives the results, when the input is gold supertags only. Since the F-score is comparable to the optimal F-score decoder, but without further fragmenting the chart, this is the decoder that is used to provide training data for the Viterbi decoder described in Chapter 4.

Table 3.4 gives the accuracy of the optimal F-score decoder on section 00, when the largest β value is used first, i.e. the adaptive supertagger setting in

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 97.39% (1863 of 1913 sentences parsed)

cats:  97.36% (41630 of 42761 tokens correct)
csent: 70.80% (1319 of 1863 sentences correct)

lp:    95.96% (36128 of 37649 labelled deps precision)
lr:    94.51% (36128 of 38225 labelled deps recall)
lf:    95.23% (labelled deps f-score)
lsent: 68.12% (1269 of 1863 labelled deps sentences correct)
```

Table 3.4: Coverage and oracle F-score on Section 00 with largest β value first

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 97.39% (1863 of 1913 sentences parsed)

cats:  98.41% (42079 of 42761 tokens correct)
csent: 81.27% (1514 of 1863 sentences correct)

lp:    97.83% (37069 of 37891 labelled deps precision)
lr:    96.98% (37069 of 38225 labelled deps recall)
lf:    97.40% (labelled deps f-score)
lsent: 79.12% (1474 of 1863 labelled deps sentences correct)
```

Table 3.5: Coverage and oracle F-score on Section 00 with smallest β value first

which the parser requests more supertags if it cannot find a spanning analysis. Table 3.5 gives the oracle results for the opposite setting, when the smallest β value is used first, and fewer supertags are requested if the chart explodes. (Appendix C gives the details of how the oracleParser was run and the various parameter settings.) Note that gold-standard supertags are not provided as input, so this is a test of the coverage of the Java C&C grammar on unseen (CCGbank) data, given automatically assigned supertagger input.

Note how much the upper bound increases when the least restrictive supertagger setting (smallest β) is used first. Of course the trade-off here is that parsing speed reduces dramatically when using the least restrictive setting. One of the benefits of the beam search parser described later is that the least restrictive setting can be used, whilst still retaining reasonable parser speeds.

Chapter 4

The Viterbi Decoder

4.1 The Features

The feature templates for the Viterbi decoder are essentially the same as those in the C&C normal-form model, defined in terms of local rule instantiations. These rules are also augmented with various combinations of words and POS tags to create lexicalised, and head dependency-style, features [8]. Note that the DP-style of the decoder means that features are restricted to be local to these rule instantiations, in order that efficient and optimal dynamic programs can be defined over the chart (e.g. the Viterbi decoder, and also the inside-outside calculations from the dependency model of C&C [8]). However, this restriction can be lifted for the (non-optimal) beam search decoder described in Chapter 5.

One question is whether features from incorrect derivations should be used (negative features), or only those seen in correct derivations (which would be any derivation producing the correct dependency structure, given the philosophy of not relying on the `pipe` derivation files from CCGbank). The potential advantage of using negative features is that these give the model more discriminating power; the downside is that they greatly increase the size of the feature set. Here we take an intermediate position by using negative features, but only from those derivations generated by the correct supertags (some of which will be incorrect). Just over 4M features are generated in this way; more details are provided in Appendix D.

4.2 Training

The training of the Viterbi decoder uses forest files: efficient encodings of the complete chart for each training sentence. These are printed to disk and then read into memory during training. In order for the training to be correct, it is important that each forest contains a correct derivation. Since the "hidden" structured perceptron is being used, there could be more than one correct derivation (i.e. any derivation producing the gold-standard set of dependencies).

iteration	1	2	3	4	5	6	7	8	9	10
F-score	86.94	87.07	87.11	87.32	87.44	87.40	87.44	87.40	87.46	87.46

Table 4.1: Accuracies for the Viterbi decoder on section 00 for different numbers of training iterations

The hidden perceptron works by running the Viterbi decoder twice for each training example: once to find the highest-scoring correct derivation, according to the current model, which acts as the gold-standard derivation for that example (in order to do the positive perceptron update); and once to find the highest-scoring derivation overall (in order to do the negative update). To enable the first decoding, each node in the forest is marked according to whether it is part of the correct derivation, using the "marking" algorithm from [8]; then Viteribi is run over just those marked derivations (of which there could be exponentially many, represented efficiently in the forest). For the second decoding, Viterbi is run over the whole forest.

In order to use as much of the training data as possible, the gold-standard dependencies are generated using the high-precision oracle decoder described in Section 3.3. In fact, the dependencies are never explicitly stored in a file; what happens instead is that the oracle decoder is run over each forest (using the CCGbank gold-standard dependencies as the target), and the optimal derivations marked, before the forests are printed to file. Hence the training is carried out using these oracle "gold standard" dependencies, rather than the dependencies in CCGbank.

4.3 Results

The accuracy of the Viterbi parser on the development data is given in Table 4.1, for each iteration of the training process. The model converges relatively quickly, in line with the results in [7]. Table 4.2, using the model after 10 iterations, compares the accuracy with C&C, with C&C results copied from [8] and [7]. Note that the coverage of the Java C&C Viterbi parser is a little lower than the coverage figures reported for C&C, so the results are not strictly comparable. However, the main role of the Viterbi parser here is to provide a baseline score for the beam search parser, so we did not try to increase the coverage by experimenting with various combinations of β and K, as was done for C&C. For the beam search decoder, an additional "skimmer" mode was implemented which returns a set of dependencies, even when a spanning analysis cannot be found, resulting in 100% coverage for that decoder (described in Chapter 5).

4.3.1 Supertagger probabilities as features

An easy way to get a slight gain in accuracy is to include the (log-) probabilities provided by the supertagger as additional features. Let p be the product of the

Model	LP	LR	LF	LSent	Cats	Cov.
Java C&C Viterbi	87.88	87.05	87.46	37.19	94.33	98.12
C&C log-linear norm-form [8]	87.17	86.30	86.73	34.99	94.05	99.06
C&C log-linear dependency [8]	88.06	86.43	87.24	35.67	94.16	99.06
C&C perceptron norm-form [7]	87.25	86.20	86.72	—	94.08	99.37

Table 4.2: Accuracies for the Viterbi decoder on section 00; results for various models in C&C are given for comparison

λ_0	LP	LR	LF	LSent	Cats
0	87.88	87.05	87.46	37.19	94.33
1	88.02	87.22	87.62	37.40	94.42
2	88.09	87.27	87.68	37.29	94.44
4	88.17	87.35	87.76	37.45	94.48
5	**88.19**	**87.38**	**87.79**	**37.51**	**94.50**
6	88.16	87.34	87.75	37.24	94.49
10	87.91	87.07	87.49	37.08	94.38

Table 4.3: Parser accuracy on section 00 with the supertagger log-probabilities as features (test time only), weighted by λ_0. Coverage is 98.12% in all cases.

probabilities of all the supertags in a derivation d, where the probabilities are provided by the mulit-tagger; then the new score for d is as follows:

$$\text{score}(d) = \lambda_0.\log(p) + \sum_i \lambda_i.f_i(d) \tag{4.1}$$

$$= \sum_j \lambda_0.\log(p_j) + \sum_i \lambda_i.f_i(d) \tag{4.2}$$

where i ranges over the existing, integer-valued features and p_j is the probability of the lexical category for the jth word. Note that $\log(p_j)$ is local to the jth word, and so these scores factor neatly over the chart, requiring no modification of the dynamic programming structure.

Rather than try and learn λ_0, the weight for the supertagger scores, here we simply try various values of λ_0 set manually. Table 4.3 shows the accuracy of the parser on section 00, for various values of λ_0. Note that, in this experiment, the supertagger scores are being used at test time only. There is code in Java C&C to include the supertagger scores in the forests, and hence used during the decoding in training, but this experiment is left for future work.

4.3.2 Oracle experiment with gold-standard supertags

One final oracle experiment with the Viterbi decoder uses gold-standard supertags as input, but the trained parser model, rather than the oracle decoder.

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 97.18% (1859 of 1913 sentences parsed)

cats:  100.00% (43565 of 43565 tokens correct)
csent: 100.00% (1859 of 1859 sentences correct)

lp:    97.89% (37885 of 38702 labelled deps precision)
lr:    97.56% (37885 of 38834 labelled deps recall)
lf:    97.72% (labelled deps f-score)
lsent: 69.39% (1290 of 1859 labelled deps sentences correct)
```

Table 4.4: Accuracy of the Viterbi decoder on Section 00 with gold-standard supertags as input

So this evaluation provides an indication of how well the parser would perform with the perfect supertagger. Table 4.4 shows that the accuracy of the parser is extremely high in this setting, again demonstrating the signifcant amount of information provided by the correct supertags. The coverage is relatively low because any sentence in Section 00 which has a gold supertag not in the lexicon (i.e. not seen in 2-21) is ignored for this evaluation, and there are also some sentences for which a spanning analysis cannot be found when only provided with a single gold-standard supertag for each word. The corresponding experiment with the beam search parser in Chapter 5 has a higher coverage figure since there we are able to use the additional "skimmer" mode.

Chapter 5

The Beam Search Decoder

The beam search idea implemented here is extremely simple: rather than store sets of non-terminals in each chart cell (with back pointers to children), partial derivations are stored instead, where each partial derivation spans the sub-sequence in the sentence corresponding to the particular cell. With a beam size of B, only the B highest-scoring partial derivations are retained in each cell. The bottom-up order of combination is the same as the Viterbi decoder, but the notion of equivalence is dropped, meaning that a) the search is no longer optimal; but b) there is no corresponding restriction on the locality of the features. Hence the chart is no longer serving as a data structure to enable dynamic programming, but simply as a means to store the partial derivations and provide a (bottom-up) order for their combination, as well as the means by which hypotheses are compared when applying the beam (since hypotheses are considered to be comparable when they are in the same cell).

The beam search applied to the chart is essentially the chart-based analogue of the beam search used in shift-reduce dependency and constituency parsers (including those for CCG) [29, 30, 26]. A similar idea was described for a chart-based dependency parser in a Cambridge NLIP seminar given by Ryan McDonald in 2014 (which the first author attended).

5.1 The coverage-enhancing "skimmer"

Java C&C has an additional "skimmer" mode for the beam search decoder which allows all sentences to receive some analysis.[1] For those sentences which do not receive a spanning analysis (i.e. when the "corner cell" of the chart is empty), the skimmer heuristically creates a fragmentary analysis by searching for the largest sub-derivations it can find, and printing out the dependencies created by those sub-derivations. The algorithm is given in Figure 5.1.

The algorithm first finds the cell containing a sub-derivation with the largest span. Within that cell it finds the highest-scoring sub-derivation, and prints out

[1]The term "skimming" is taken from [21].

```
Skimmer(SENTENCE):

if length(SENTENCE) == 1 then
  return

CELL = non-empty cell with largest span in SENTENCE

D_MAX = highest-scoring sub-derivation in CELL

print_dependencies(D_MAX)

if SENTENCE to left of cell is non-empty
  Skimmer(SENTENCE to left of cell)

if SENTENCE to right of cell is non-empty
  Skimmer(SENTENCE to right of cell)
```

Figure 5.1: Algorithm for the heuristic skimmer

the dependencies created by that sub-derivation. If there is any part of the input sentence remaining to the left of the cell, it recursively calls the skimmer on that remaining part; likewise for any part of the input sentence to the right of the cell.

The skimmer is only called at test time, and not used during training.

5.2 Training

It is possible to adopt the hidden-variable perceptron used for the Viterbi decoder directly for the beam search decoder: run the beam search decoder on each training sentence, and extract two derivations from the root cell in the chart for the perceptron update. The gold derivation (for the positive update) is the highest-scoring derivation in the root cell which produces the correct dependencies, and the incorrect derivation (for the negative update) is the highest-scoring derivation from all derivations in the root cell. However, there is a potential problem when applying this scheme to the beam search decoder, which is that all gold derivations may have been discarded by the beam, so there is no correct derivation in the root cell to use for the positive update.

The max-violation framework of Huang et al. (2012) [18] provides a solution. In this perceptron training framework (applied to the chart parser), the updates do not have to occur in the root cell. Rather, the updates are based on the cell in which the *maximum violation* occurs, i.e. the cell where the difference in score between the highest-scoring sub-derivation and the highest-scoring correct sub-derivation is the greatest. Intuitively, this is the place in the chart where the decoder is making the largest error. Hence, if there are no correct derivations in the root cell, it does not matter since an update can still occur elsewhere in the chart (where there are correct sub-derivations).

There is an additional subtlety when applying max-violation in the hidden-variable case, when modelling dependency structures rather than normal-form derivations. Consider an arbitrary cell in the chart: how can we determine whether a sub-derivation in that cell is correct? The sub-derivation must produce the correct dependencies for the corresponding sub-sequence in the sentence, but how can those dependencies be determined, since the training data contains the set of correct dependencies for the whole sentence?

The solution adopted here is as follows. The oracle decoder described in Section 3.3 is used to create a packed chart, and the nodes in each correct derivation are marked, using the "marking" algorithm from [8]. (This algorithm has already been used in Section 3.3 when creating the forest files.) Then, for each cell in the chart, any correct sub-derivation in that cell is chosen and the corresponding dependencies are taken as the gold dependencies for that cell. Note that the gold dependencies are from the complete sub-derivation, i.e. spanning the complete sub-sequence of the sentence for the cell, and not just dependencies produced at that point in the chart. We make the (reasonable) assumption that all correct sub-derivations in a cell produce the same subset of gold dependencies.

Now there are subsets of gold dependencies at various places in the chart, and not just in the root cell, and so the max-violation update scheme can be applied. Note that some cells may not contain gold dependencies, either because there are no correct sub-derivations rooted in that cell, or because any correct sub-derivations rooted there have not produced any dependencies. Such cells are not used for updates.

Darren Foong's Part II dissertation [15] explored a number of max-violation update strategies, including one where more than cell can be used for the update, based on the intuition that the partial derivations used for the update should span the whole sentence [27]. The results in this report are based only on the single max-violation for each sentence. Darren's disseration also explored the use of cube pruning [28], in this case as a further heuristic technique for increasing the efficiency of the parser. The results in this report did not use cube pruning.

5.3 Results

5.3.1 Viterbi-trained model with local features

The first experiment simply applies the beam with the Viterbi-trained model, using the same local features as C&C. Table 5.1 gives the results, with a beam size of 32. One interesting feature of the beam search decoder is that using the (log) supertagger probabilities as an extra real-valued feature makes a huge improvement, more so than in the Viterbi case (see Table 5.2). This is presumably because, unlike the Viterbi decoder, the beam search decoder is unable to

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 99.16% (1897 of 1913 sentences parsed)

cats:  93.38% (41821 of 44787 tokens correct)
csent: 43.07% (817 of 1897 sentences correct)

lp:    86.29% (34161 of 39587 labelled deps precision)
lr:    85.67% (34161 of 39874 labelled deps recall)
lf:    85.98% (labelled deps f-score)
lsent: 35.37% (671 of 1897 labelled deps sentences correct)
```

Table 5.1: Accuracy of the beam search decoder on Section 00, with a beam size of 32, using the Viterbi-trained model

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 99.22% (1898 of 1913 sentences parsed)

cats:  94.28% (42247 of 44808 tokens correct)
csent: 48.47% (920 of 1898 sentences correct)

lp:    87.80% (34738 of 39566 labelled deps precision)
lr:    87.08% (34738 of 39893 labelled deps recall)
lf:    87.44% (labelled deps f-score)
lsent: 37.88% (719 of 1898 labelled deps sentences correct)
```

Table 5.2: Accuracy of the beam search decoder on Section 00, with a beam size of 32 and $\lambda = 5$, using the Viterbi-trained model

recover from any poor choices made low down in the chart, and selecting the correct lexical categories is difficult given only the local features.

Note that the coverage for the beam search parser is higher than the corresponding Viterbi decoder results (Table 4.3), so the results of the two decoders are not strictly comparable. However, the beam search parser with the supertagger probability feature is extremely competitive (perhaps surprisingly so). One of the reasons is that the beam search parser is given multitagger input with β set to 0.01, whereas the first β value under the adaptive multitagging strategy for the Viterbi decoder was 0.75, meaning that more of the search space is at least available for the beam search parser to explore.

Table 5.3 gives the results with the skimmer enabled, so that the coverage is now 100%. The overall LF only reduces by 0.32%, which is encouraging given that the additional sentences now parsed are likely to be some of the longer, more difficult sentences. Table 5.4 shows the effect of beam size on the accuracy,

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 100.00% (1913 of 1913 sentences parsed)

cats:  94.08% (42732 of 45422 tokens correct)
csent: 48.20% (922 of 1913 sentences correct)

lp:    87.59% (35015 of 39978 labelled deps precision)
lr:    86.66% (35015 of 40405 labelled deps recall)
lf:    87.12% (labelled deps f-score)
lsent: 37.58% (719 of 1913 labelled deps sentences correct)
```

Table 5.3: Accuracy of the beam search decoder on Section 00, with a beam size of 32 and $\lambda = 5$, and the skimmer enabled, using the Viterbi-trained model

Beam size	LP	LR	LF	Cats	% skim
4	83.37	77.46	80.31	92.07	30.89
8	87.10	85.86	86.48	93.85	1.88
16	87.45	86.43	86.94	94.01	1.05
32	87.59	86.66	87.12	94.08	0.78
64	87.63	86.74	87.18	94.08	0.78

Table 5.4: Accuracy of the beam search decoder on Section 00, with varying beam sizes, $\lambda = 5$, and the skimmer enabled, using the Viterbi-trained model; % skim is the percentage of test sentences parsed using the skimmer

indicating that little gains can be had beyond a beam size of 32 (at least with this model).

Table 5.6 gives the results of the oracle experiment using the gold-standard supertags as input on section 00, but with the beam search decoder (and skimmer enabled[2]). As for the Viterbi decoder, the overall accuracy is extremely high, showing again what could be achieved with a perfect supertagger.

5.3.2 Max-violation-trained model with non-local features

One of the main motivations for the beam search decoder is that it does not impose any restrictions on the locality of the features. The non-local features we have explored are variants on the "grandparent" features used in reranking models [5]. Darren Foong's dissertation [15] describes the features in detail.

Table 5.7 gives the results. This model was trained using max-violation, for 14 iterations, with the weights for the local features initialised with the values from the Viterbi training (and the weights for the non-local features initialised

[2]The coverage is still not 100% because of those sentences in section 00 with gold supertags not in the lexicon.

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 97.33% (1862 of 1913 sentences parsed)

cats:  100.00% (43907 of 43907 tokens correct)
csent: 100.00% (1862 of 1862 sentences correct)

lp:    97.80% (38079 of 38936 labelled deps precision)
lr:    97.43% (38079 of 39082 labelled deps recall)
lf:    97.62% (labelled deps f-score)
lsent: 69.07% (1286 of 1862 labelled deps sentences correct)
```

Table 5.5: Accuracy of the beam search decoder on Section 00 with gold-standard supertags as input, with a beam size of 32, using the Viterbi-trained model

at zero). Note that the accuracy has now surpassed that of the Viterbi decoder, with a higher coverage. Table 5.8 gives the results with the skimmer enabled, so that coverage is 100%.

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 99.11% (1896 of 1913 sentences parsed)

cats:  100.00% (44688 of 44688 tokens correct)
csent: 100.00% (1896 of 1896 sentences correct)

lp:    97.80% (38667 of 39537 labelled deps precision)
lr:    97.19% (38667 of 39783 labelled deps recall)
lf:    97.50% (labelled deps f-score)
lsent: 67.83% (1286 of 1896 labelled deps sentences correct)
```

Table 5.6: Accuracy of the beam search decoder on Section 00 with gold-standard supertags as input, with a beam size of 32 and the skimmer enabled, using the Viterbi-trained model

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 99.06% (1895 of 1913 sentences parsed)

cats:  94.59% (42079 of 44488 tokens correct)
csent: 50.61% (959 of 1895 sentences correct)

lp:    88.32% (34898 of 39515 labelled deps precision)
lr:    87.89% (34898 of 39706 labelled deps recall)
lf:    88.10% (labelled deps f-score)
lsent: 40.79% (773 of 1895 labelled deps sentences correct)
```

Table 5.7: Accuracy of the beam search decoder on Section 00, with a beam size of 32, using the larger feature set and the max-violation-trained model (14 iterations)

```
note: all these statistics are over just those sentences
      for which the parser returned an analysis

cover: 100.00% (1913 of 1913 sentences parsed)

cats:  94.25% (42812 of 45422 tokens correct)
csent: 50.24% (961 of 1913 sentences correct)

lp:    88.00% (35273 of 40082 labelled deps precision)
lr:    87.30% (35273 of 40405 labelled deps recall)
lf:    87.65% (labelled deps f-score)
lsent: 40.41% (773 of 1913 labelled deps sentences correct)
```

Table 5.8: Accuracy of the beam search decoder on Section 00, with a beam size of 32, and the skimmer enabled, using the larger feature set and the max-violation-trained model (14 iterations)

Appendix A

CCGbank Data Preprocessing

First extract the data from CCGbank, in the appropriate format, using the `scripts/create_data` script. This is a truncated version of the `create_data` script from the C&C repository. Note that it relies on a number of scripts from the C&C release, and so is run here from the C&C directory:

```
squacco:candc-1.00$ pwd
/home/sc609/parser/candc-1.00
squacco:candc-1.00$ ~/code/java-candc/scripts/create_data \
  ~/data/CCGbank/CCGbank1.2 ~/code/java-candc/data
converting AUTO files to old .pipe format
creating CCGbank PARG files (grouped by section)
extracting CCGbank POS/super-tagged text
extracting CCGbank category lists
converting PARG files into our dependency format
```

Now various files have been created in the `data/gold` directory, principally raw, pos tag, supertag, and dependency files for the training, development and test sections.

Appendix B

Training the Supertagger

One difference with the C&C supertagger is that we can now use all $\approx$ 1,200 lexical categories, and so the supertag frequency cutoff needs setting on the command line (to zero; see below). The supertagger training and testing was run on a MacBook Pro, using version 1.0 of the C&C tools and the same data created as in Appendix A. This was the command used to train the supertagger on all the training data for supertagging unseen data:

```
unknown-f8:1e:df:d9:78:a9-2:candc-1.00 stephenclark$ bin/train_super \
--verbose --comment "using all lexical categories" --input \
~/Mystuff/code/java/java_parser_repos/baseline_expts/working/gold/wsj02-21.stagged \
--model ~/Mystuff/code/java/java_parser_repos/baseline_expts/models/super \
--super-category_cutoff 0
```

Jacknifing the supertagger training data The supertagger needs training on N-1 separate chunks of the supertagger training data, so that it can be used to supertag the remaining chunk. N = 10 is a sensible number of chunks. This process needs to happen N times, with the N supertagged chunks concatenated together to give the supertagged training data (i.e. data for input to the parser training process). When supertagging the remaining chunk, use the `msuper` program with the same values of β and K which will be used at test time. If the adapative supertagging method is being used at test time (i.e. potentially using various values of β), choose values for β and K which will cover the least restrictive case. For the experiments reported here, $\beta = 0.01$ and $K = 100$.

Sections 2-21 were split into 10 chunks:

```
squacco:jacknifed$ wc -l wsj02-21.*
     3961 wsj02-21.pos.1
     3955 wsj02-21.pos.10
     3961 wsj02-21.pos.2
     3961 wsj02-21.pos.3
     3961 wsj02-21.pos.4
     3961 wsj02-21.pos.5
     3961 wsj02-21.pos.6
     3961 wsj02-21.pos.7
     3961 wsj02-21.pos.8
     3961 wsj02-21.pos.9
```

```
    3961 wsj02-21.stagged.1
    3955 wsj02-21.stagged.10
    3961 wsj02-21.stagged.2
    3961 wsj02-21.stagged.3
    3961 wsj02-21.stagged.4
    3961 wsj02-21.stagged.5
    3961 wsj02-21.stagged.6
    3961 wsj02-21.stagged.7
    3961 wsj02-21.stagged.8
    3961 wsj02-21.stagged.9
   35643 wsj02-21.stagged.comb.1
   35649 wsj02-21.stagged.comb.10
   35643 wsj02-21.stagged.comb.2
   35643 wsj02-21.stagged.comb.3
   35643 wsj02-21.stagged.comb.4
   35643 wsj02-21.stagged.comb.5
   35643 wsj02-21.stagged.comb.6
   35643 wsj02-21.stagged.comb.7
   35643 wsj02-21.stagged.comb.8
   35643 wsj02-21.stagged.comb.9
  435644 total
```

For example, wsj02-21.stagged.1 is the first chunk of 3961 supertagged sentences from sections 2-21 of CCGbank. The files used for training are the .comb files. For example, wsj02-21.stagged.comb.1 is a concatentation of all the files *excluding* wsj02-21.stagged.1. The resulting model will be used to supertag wsj02-21.pos.1, which will then become part of the input for creating the training data.

This is how the model used to supertag the first chunk was trained:

```
bin/train_super --model \
~/Mystuff/code/java/java_parser_repos/models/super.1 \
--comment "trained on splits 2-10 of 2-21" --input \
~/Mystuff/code/java/java_parser_repos/data/jacknifed/wsj02-21.stagged.comb.1 \
--verbose --super-category_cutoff 0
```

And this how the model is used to supertag that chunk:

```
bin/msuper --model ~/Mystuff/code/java/java_parser_repos/models/super.1
--dict_cutoff 100 --beta 0.01 --input
~/Mystuff/code/java/java_parser_repos/data/jacknifed/wsj02-21.pos.1 --output
~/Mystuff/code/java/java_parser_repos/working/jack_knifed_stagged/wsj02-21.stagged.0.01.100.1
```

The accuracy of the multitagger on the various splits was given in Table 2.1, along with the single-tag supertagger accuracy (although the single-tag data is never used by the parser). These accuracies were calculated as follows (after transferring the data from the laptop to the local repository):[3]

```
squacco:java-candc$ scripts/evaluateTagger.pl data/jacknifed-gold/wsj02-21.stagged.1 \
  data/auto-stagged/jacknifed/wsj02-21.stagged.1
86376 correct out of 93036 = 92.8414807171417 %
incorrect: 6660
```

[3]The data is not part of the public release since it is covered by the CCGbank licence.

```
sentences totally correct 1422 out of 3961 = 35.90002524615 %

squacco:java-candc$ scripts/evaluateTaggerKbest.pl data/jacknifed-gold/wsj02-21.stagged.1 \
  data/auto-stagged/jacknifed/wsj02-21.stagged.0.01.100.1
91885 correct out of 93036 = 98.7628444903048 %
1.89227825787867 average number of tags per word
sentences totally correct 3203 out of 3961 = 80.8634183287049 %
```

Appendix C

This Appendix describes some of the main components of the grammar, the corresponding files of which reside in the `grammar` directory. It also describes how the oracle experiments were carried out in Section 3.3.

The Type-Raising files

The four files containing instances of type-raising for four categories — `trNP`, `trPP`, `trAP`, `trVP_to` — were copied from the C&C source.

Ignoring some Dependencies for Evaluation

The following files are used by the OracleDecoder when calculating maximum F-scores, since it needs to ignore relations which are ignored by the evaluate script (i.e. those produced by the parser but not in CCGbank):

```
relsHeadsFillersNoEval.txt relsHeadsNoEval.txt relsNoEval.txt ruleIDsNoEval.txt
```

These files were all created by hand by examining the evaluate script.

The markedup File

This section was written by Luana Bulat.

The markedup file is the file containing the lexical categories, together with annotation which determines dependency and head information to be used by the parser. The file uses the following format:

```
<lexical_category>
<#_of_argument_slots> <markedup_category>
```

Consider the lexical category for a transitive verb, $(S[dcl]\backslash NP)/NP$. The lines in the markedup file corresponding to this entry are as follows:

```
(S[dcl]\NP)/NP
2 ((S[dcl]{_}\NP{Y}<1>){_}/NP{Z}<2>){_}
```

Variables in curly brackets represent head information and the ones in angle brackets dependency relations. For example, the above line indicates that in the transitive verb category, there are two dependency relations (one between the verb and its object and one between the verb and its subject).

Extending the lexicon

The markedup file in the C&C parser was obtained by manually annotating the 580 most frequent CCGbank categories. A desired feature of the Java C&C parser was to extend its lexicon to cover all the 1285 categories contained in CCBank. Manually annotating the remaining categories would have been unfeasible, since the less frequent CCG categories tend to get very long, making the annotation slow and error-prone.

Algorithm for markedup creation

Automatically augmenting lexical entries with head and dependency information is a straightforward process in the case of local dependencies and can be done recursively as follows:

- If the category is atomic, then just add head information
  Example. `NP` becomes `NP{_}`; `S[dcl]` becomes `S[dcl]{_}`
- If the category is complex of type `X|X`, the assigned markedup category will be `(X'{Y}|X'{Y}<1>){_}`, where:
  - `X'` is the markedup category for `X` with the last `{_}` removed and without any argument slots
  - `Y` is the first unused variable in `X'`

  Example. `NP\NP` becomes `(NP{Y}\NP{Y}<1>){_}`
- If the category is complex of type `X|Y`, the assigned markedup category will be `(X'|Y'){_}`, where:
  - `X'` is the markedup category for `X`
  - `Y'` is the markedup category for `Y`, modified so that no variables clash with the ones in `X'`

  Example. `(S[dcl]\NP)/PP` becomes `((S[dcl]{_}\NP{Y}<1>){_}/PP{Z}<2>){_}`

Annotating non-local dependencies using co-indexation

However, creating the markedup category for cases in which non-local dependencies are needed is not as straightforward. For example, consider the case of the relative pronoun *that*, with the corresponding lexical category `(NP\NP) / (S[dcl]/NP)`. In the context of parsing the phrase *movie that Mary loves*, we need to ensure that a dependency is created between *movie* and *loves*, meaning

that we want to specify that the head of the `NP` argument of the relative clause be the same as the head of the `NP` it modifies. The correct markedup category for *that* is listed below:
`((NP{Y}\NP{Y}<1>){_}/(S[dcl]{Z}/NP{Y}){Z}<2>){_}`.

Fortunately, the CCGbank manual contains a list of 237 categories that contain non-local dependencies (henceforth special categories), together with their correct markedup annotation. We have incorporated these as special cases in our algorithm by simply retrieving the correct markedup from the CCGbank manual, instead of performing the recursive function call.

Creating final markedup file

We have tested our algorithm by running it on the 580 categories contained in the C&C markedup file and comparing its output with the manual annotations. This allowed for careful inspections of the special cases, as well as the discovery of 29 annotation errors in the C&C markedup file. Once all the conflicting cases had been analysed and solved, the algorithm was run on the full CCG lexicon.

The Rule Instances

The `all_rule_instances` file is used in conjunction with the `seenRules` parameter, and obtained with the `extract_rules` script from C&C. However, there is a bug in the 1.0 release, so it is important to use the version of the script in the SVN repository. The file obtained from the `extract_rules` script has then been further processed to create additional rule instances, by replacing all instances of NP with NP[nb], since the [nb] feature needs to be part of the comparison (given how the categories are compared in Java C&C). The grammar directory also contains the original `all_rule_instances_candc_SVN` file before the processing. The final `all_rule_instances` file used here has 3,248 instances.

The Oracle Experiments

Data preprocessing The `OracleParser` program requires the gold supertagged input data in two formats: one where each word is on a separate line, with spaces separating word, POS tag and supertag; and one where, in addition, there is a 1 before each supertag (signifying the number of supertags) and a 1 at the end of each line (signifying the probability of the supertag). The former is to provide gold supertags when augmenting automatic supertagger output, and the latter when the gold supertags are the only input. The `reformat_stagged.pl` script performs this reformatting, with an argument at the top of the script which determines whether the 1s are printed.

```
squacco:java-candc$ scripts/reformat_stagged.pl < data/gold/wsj02-21.stagged \
  > data/gold/wsj02-21.stagged.reformat
squacco:java-candc$ scripts/reformat_stagged.pl < data/gold/wsj02-21.stagged \
  > data/gold/wsj02-21.stagged.msuper
```

The `OracleParser` program also requires gold-standard dependencies, in a different format to the dependency files used for evaluation (with the words replaced by word indices). The following script performs the conversion (just taking out the words and leaving in the indices, which are already there):

```
squacco:java-candc$ scripts/remove_words.pl data/gold/wsj02-21.ccgbank_deps \
  > data/gold/wsj02-21.gold_deps
```

Running the OracleParser This is how the oracle parser was run with just gold-standard supertags as input, to give the results in Table 3.1:

```
java -classpath bin OracleParser data/gold/wsj02-21.stagged.msuper \
  data/gold/wsj02-21.stagged.reformat working/output/oracle.2-21.out \
  working/output/oracle.2-21.log data/gold/wsj02-21.gold_deps null 1 40000
```

The null argument is for an optional roots file, where the oracle decoder can be constrained to only find derivations rooted in a particular category. The parameters in the OracleDecoder were as follows:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 1000000;
boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = true; // features don't get read in by the OracleParser

boolean depsSumDecoder = false;
// if false then uses the F-score decoder
boolean oracleFscore = !depsSumDecoder;
// F-score decoder is calculating a true oracle score

String grammarDir = "grammar";

RuleInstancesParams ruleInstancesParams = new RuleInstancesParams(true, false, \
  false, false, false, false, grammarDir); \\ first true is all seen_rules
```

Note that both the `seen_rules` and `eisnerNormalForm` parameters are set to true. These were switched off to see if the coverage, and F-score, could be increased, but they appear to have little effect, on this oracle test at least. Even if the upper bound on accuracy is increased, coverage is adversely affected since there are now more chart "explosions" (i.e. where the maximum number of categories in the chart is exceeded, here set to 1,000,000).

This is how the oracle parser was run with the gold-standard supertags as augmented with additional supertags from the supertagger, to give the results in Table 3.2:

```
java -classpath bin OracleParser \
  data/auto-stagged/jacknifed/wsj02-21.stagged.0.01.100.all \
  data/gold/wsj02-21.stagged.reformat \
  working/output/oracle.2-21.augmented.out \
  working/output/oracle.2-21.augmented.log \
  data/gold/wsj02-21.gold_deps null 1 40000
```

The parameters in the OracleParser file were the same as above, except `MAX_SUPERCATS` was increased to 2,000,000, and the series of β values was set as follows:

```
boolean adaptiveSupertagging = true;
double[] betas = { 1.0, 0.075, 0.03, 0.01, 0.001, 0.0001 };
```

The idea is that the oracle parser should first try to parse with just the gold supertags, and only add additional supertags (with decreasing β values) if a spanning analysis cannot be found.

The `OracleDepsSumDecoder` program is run in a similar way. Currently it also requires the `roots` file as an argument, although it is not clear that constraining the derivations in this way makes any difference. The `data/gold/wsj02-21.roots` file was obtained by running `scripts/extract_roots.pl` on `data/gold/wsj02-21.pipe`.

```
squacco:java-candc$ java -classpath bin OracleParser \
  data/gold/wsj02-21.stagged.msuper null \
  data/oracle-gold/wsj02-21.oracle_gold_deps \
  data/oracle-gold/wsj02-21.oracle_gold_deps.log \
  data/gold/wsj02-21.gold_deps data/gold/wsj02-21.roots 1 40000
```

For the experiments on the development test data, without gold-standard supertags as input, the following command was run:

```
java -classpath bin OracleParser data/auto-stagged/wsj00.stagged.0.01.100 null \
  /local/filespace/sc609/oracle.00.out /local/filespace/sc609/oracle.00.log \
  data/gold/wsj00.gold_deps null 1 2000
```

Two separate results were obtained, one where the largest β value is used first:

```
boolean adaptiveSupertagging = true;
double[] betas = { 0.075, 0.03, 0.01, 0.001 };
```

and another where the smallest β value is used first:

```
boolean adaptiveSupertagging = false;
double[] betas = { 0.01, 0.01, 0.01, 0.03, 0.075 };
```

The other settings in the `OracleParser` program were as follows:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 2000000;
boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = false; // features don't get read in by the OracleParser

boolean depsSumDecoder = false;
// if false then uses the F-score decoder
boolean oracleFscore = !depsSumDecoder; // F-score decoder is calculating a true oracle score

String grammarDir = "grammar";

RuleInstancesParams ruleInstancesParams = \
  new RuleInstancesParams(true, false, false, false, false, false, grammarDir);
```

The `OracleParser` program can be run to produce output for evaluation, in order to create the results tables in Chapter 3, or used to produce dependencies for training, for the beam search decoder. (For the Viterbi decoder, the training data is produced with the `PrintForests` program.) More detail on the training data for the beam search decoder is given in Appendix E.

Appendix D

The words_feats Directory

The parser, when creating features, internally represents words and postags as integers, which are obtained by reading in a file which lists all the words and postags in the training data. That file is obtained as follows:

```
squacco:java-candc$ scripts/extract_word_pos.pl < \
  data/gold/wsj02-21.stagged | sort | uniq > words_feats/wsj02-21.wordsPos
```

Generating the Features File

The model features used by the parser need generating beforehand using the `CountFeatures` program. Section 4.1 explained the decision to use negative features, but only those generated by derivations arising from the gold-standard lexical categories. So the input to `CountFeatures` is just the gold supertags, but the program counts features from every derivation in the chart. The program is run as follows:

```
java -classpath bin CountFeatures \
  data/gold/wsj02-21.stagged.msuper words_feats/wsj02-21.feats.1-22 \
  words_feats/wsj02-21.feats.log 1 40000
```

The 1-22 suffix indicates the feature types extracted (which are listed in the `src/model/Features` class). The number of features extracted using these types and this input is just over 4 million:

```
squacco:java-candc$ grep -v '^# ' words_feats/wsj02-21.feats.1-22 | \
  grep -v '^$' | wc -l
4071508
```

Here are the settings in `src/CountFeatures.java`:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 1000000;

boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = false;
boolean oracleFscore = false;

String grammarDir = "grammar";
String lexiconFile = "words_feats/wsj02-21.wordsPos";
String featuresFile = "words_feats/wsj02-21.feats.1-22"; // the output file

RuleInstancesParams ruleInstancesParams = \
  new RuleInstancesParams(true, false, false, false, false, false, grammarDir);

boolean adaptiveSupertagging = true;
double[] betas = { 0.01 }; // assume passing in gold-standard supertags only,
                           // so only one beta value needed
```

The number of sentences from which the features are extracted is 38,868, which represents a coverage figure of 98.1% of the training data. 1.8% of sentences did not receive a spanning analysis, and 0.1% exceeded the maximum number of categories.

Generating the Forest Files

The PrintForests program uses the jacknifed supertagger input to create packed charts/forests for each training sentence, which are printed to disk using an efficient encoding. PrintForests also runs the oracle decoder, marking nodes in the forests corresponding to correct derivations (which is why one of the arguments is the gold_deps file).

PrintForests takes two integer arguments – start and end sentence numbers – which allow it to parse contiguous subsets of the data. This allows the printing of the forests to be run in parallel. Here the forests are being created in five chunks, each chunk corresponding to 8,000 training sentences (except the last one, with marginally fewer sentences, since there are 39,604 in total); this command creates the third chunk:

```
java -Xss10m -classpath bin PrintForests \
  data/auto-stagged/jacknifed/wsj02-21.stagged.0.01.100.all \
  data/gold/wsj02-21.stagged.reformat /local/filespace/sc609/forests.out.3 \
  /local/filespace/sc609/forests.log.3 data/gold/wsj02-21.gold_deps \
  data/gold/wsj02-21.roots 16001 24000 &
```

Note the `-Xss10m` command line option, which prevents a stack overflow error.

The separate forest files then need merging into one, using the following script:

```
squacco:java-candc$ scripts/merge_forests.pl /local/filespace/sc609/forests.out 5 \
  > /local/filespace/sc609/forests.out.all
```

The coverage of the complete forest file is:

```
squacco:sc609$ grep '^$' forests.out.all | wc
  39294       0   39294
```

which corresponds to 39,294 / 39,604 = 99.2% of the training data being used for training.

The Viterbi Training Program

There are two versions of the training program: `TrainViterbiAllInMemory` and `TrainViterbi`. Both produce the same model; the difference is that, in the latter, each forest is read into memory one at a time and then discarded (or at least potentially discarded, depending on the automatic garbage collection process); whereas in the former all forests are forced to be held in memory at once. Given that `TrainViterbi` finishes in a reasonable time, and uses substantially less memory, this is the recommended version to use. The training program is run as follows, where 10 is the number of passes through the training data:

```
squacco:java-candc$ java -classpath bin TrainViterbi \
  /local/filespace/sc609/forests.out.all \
  /local/filespace/sc609/weights.viterbi 10 &
```

Supertagger Probabilities as Features

These were the settings used to obtain the results in Section 4.3.1:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 1000000;

boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = false;
boolean oracleFscore = false;
...
boolean adaptiveSupertagging = true;
double[] betas = { 0.075, 0.03, 0.01, 0.001 };
...
double lambda = 5.0;

java -classpath bin Parser data/auto-stagged/wsj00.stagged.0.01.100 \
  /local/filespace/sc609/parser.viterbi.lambda=5.out \
  /local/filespace/sc609/parser.viterbi.lambda=5.log \
  /local/filespace/sc609/weights 1 2000 > /dev/null &
```

Parser Evaluation

The various accuracy tables given in this report, for example Table 5.4, are created using the `evaluate_new` script:

```
squacco:java-candc$ scripts/evaluate_new data/gold/wsj00.stagged \
  data/gold/wsj00.ccgbank_deps /local/filespace/sc609/parser.out
```

The `evaluate_new` script is essentially the same as `evaluate` from C&C, except it has been modified to handle minor changes in how arguments are encoded in the markedup file compared to C&C.

Appendix E

Generating Oracle Dependencies for Beam Search Training

Unlike the Viterbi training, which effectively embeds the oracle dependencies used for training in the forest files, the beam search training requires the oracle dependencies to be explicitly printed. This is how the oracleParser was run to generate the oracle dependencies for beam search training:

```
java -classpath bin OracleParser data/auto-stagged/jacknifed/wsj02-21.stagged.0.01.100.all \
  data/gold/wsj02-21.stagged.reformat /local/filespace/sc609/training_oracle_deps.02-21 \
  /local/filespace/sc609/training_oracle_deps.02-21.log data/gold/wsj02-21.gold_deps \
  data/gold/wsj02-21.roots 1 40000
```

And these were the settings used in OracleParser:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 2000000;
boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = false; // features don't get read in by the OracleParser

boolean depsSumDecoder = false;
// if false then uses the F-score decoder
boolean oracleFscore = !depsSumDecoder; // F-score decoder is calculating a true oracle score

String grammarDir = "grammar";

RuleInstancesParams ruleInstancesParams = \
  new RuleInstancesParams(true, false, false, false, false, false, grammarDir);
// first boolean argument says use all rule instances; the rest are subsets (eg just punct)

...

boolean adaptiveSupertagging = false;
double[] betas = { 0.01, 0.01, 0.01, 0.03, 0.075 };

boolean training = true;
// set to false if the string category and slot are needed for evaluation;
// true gives deps used for training
```

This is how the beam search parser was run to get the results in Table 5.1, which uses the Viterbi-trained model with local features, but with beam search employed instead of dynamic programming:

```
squacco:java-candc$ java -classpath bin ParserBeam data/auto-stagged/wsj00.stagged.0.01.100 \
  /local/filespace/sc609/parser.beam.32.lambda\=0.out \
  /local/filespace/sc609/parser.beam.32.lambda\=0.log \
  /local/filespace/sc609/weights 1 2000
```

And these were the various parameter settings:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 500000;

boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = false;
boolean compactWeights = true;
boolean cubePruning = false;

String grammarDir = "grammar";
String lexiconFile = "words_feats/wsj02-21.wordsPos";
String featuresFile = "words_feats/wsj02-21.feats.1-22";

RuleInstancesParams ruleInstancesParams = \
  new RuleInstancesParams(true, false, false, false, false, false, grammarDir);

double[] betas = { 0.0001 };
// just one beta value needed - no adaptive supertagging

int beamSize = 32;
double beta = Double.NEGATIVE_INFINITY;
// this is the beta used by the parser, in a second beam
// the closer to zero the more aggressive the beam

boolean applySkimmer = false;
...
double lambda = 0.0; // weights log probs from the supertagger; 0.0 ignores these probs
```

The beam search parser is able to use additional (non-local) features, and so a number of additional feature templates have been designed based on the notion of "grandparent" features. In order to generate those features, the `CountFeaturesBeam` program is used, which is a variant on the `CountFeatures` program described earlier and used in conjunction with the Viterbi decoder. In fact, the same code is used to recurse down from the root of the derivations spanning the sentence, counting features. The difference is that, in the beam search case, only B derivations are used (where B is the size of the beam).[4] Another difference is that, in the Viterbi case, the input to `CountFeatures` was gold-standard supertags. Here the jacknifed input is used, with $\beta = 0.01$, since this will produce features closer to those seen at test time. The jacknifed input

[4]An alternative, which would generate more features, would be to call `CountFeaturesDecoder` on every cell in the beam-search chart, not just the root cell.

can be used without producing too many features since `CountFeaturesDecoder` is being run on exponentially fewer derivations in the beam search case.

This was the command used to produce the expanded feature set, with the parser settings below; note the use of $\lambda = 5$, again mirroring the situation at test time:

```
java -classpath bin CountFeaturesBeam
  data/auto-stagged/jacknifed/wsj02-21.stagged.0.01.100.all \
  words_feats/wsj02-21.feats.all.lambda=5 \
  /local/filespace/sc609/weights.viterbi_init.zeros.all \
  /local/filespace/sc609/count_feats_beam.log /local/filespace/sc609/weights.viterbi 1 40000

int MAX_WORDS = 250;
int MAX_SUPERCATS = 50000;

boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = false;
boolean newFeatures = true;
boolean cubePruning = false;
boolean oracleFscore = false;

String grammarDir = "grammar";
String lexiconFile = "words_feats/wsj02-21.wordsPos";
String featuresFile = "words_feats/wsj02-21.feats.1-22";

RuleInstancesParams ruleInstancesParams = new \
  RuleInstancesParams(true, false, false, false, false, false, grammarDir);

double[] betas = { 0.0001 };
// just one beta value needed - no adaptive supertagging

int beamSize = 32;
double lambda = 5.0;
```

The number of features in the expanded feature set is as follows, compared with the local Viterbi feature set (although note that each file has 3 preface lines at the top, so the number of features is in fact 4071508 and 8137733):

```
goldeneye:words\_feats$ wc wsj02-21.feats.all.lambda\=5 wsj02-21.feats.1-22
  8137736  57507559 375716141 wsj02-21.feats.all.lambda=5
  4071511  27705530 154964900 wsj02-21.feats.1-22
```

This is the command used to train the beam search parser; note the initial weights file is the one output from the `CountFeaturesBeam` program used above (weights.viterbi_init.zeros.all):[5]

```
sc609@goldeneye:~/code/java-candc$ java -classpath bin TrainParserBeam \
  data/auto-stagged/jacknifed/wsj02-21.stagged.0.01.100.all \
```

[5]There is a horrible inconsistency in the current code, which is that readFeatures in model/Features.java assumes an extra feature for the log probability of the lexical categories, which means each weights file has to have an extra weight at the top of the file. `CountFeaturesBeam` does not currently print the extra weight, so it needs adding manually into weights.viterbi_init.zeros.all. Future versions should have this inconsistency fixed.

```
  /local/filespace/sc609/weights.beam_train.viterbi_init.lambda=5.all \
  /local/filespace/sc609/beam_train.viterbi_init.lambda=5.all.log \
  /local/filespace/sc609/weights.viterbi_init.zeros.all \
  data/oracle-gold/wsj02-21.oracle_gold_deps_for_train 20 1 40000
```

These were the parser settings for the training:

```
int MAX_WORDS = 250;
int MAX_SUPERCATS = 50000;

boolean altMarkedup = false;
boolean eisnerNormalForm = true;
boolean detailedOutput = true;
boolean newFeatures = true;
boolean cubePruning = false;
boolean parallelUpdate = false;
boolean updateLogP = false;

String grammarDir = "grammar";
String lexiconFile = "words_feats/wsj02-21.wordsPos";
String featuresFile = "words_feats/wsj02-21.feats.all.lambda=5";

RuleInstancesParams ruleInstancesParams = new \
  RuleInstancesParams(true, false, false, false, false, false, grammarDir);

double[] betas = { 0.0001 };
// just one beta value needed - no adaptive supertagging

int beamSize = 32;
double beta = Double.NEGATIVE_INFINITY;
// this is the beta used by the parser, in a second beam
// the closer to zero the more aggressive the beam

double lambda = 5.0;
```

At test time the default parameter settings were used (in src/io/Params.java), but with MAX_SUPERCATS increased to 100000.

Bibliography

[1] Michael Auli. *Integrated Supertagging and Parsing.* PhD thesis, University of Edinburgh, 2012.

[2] Johan Bos. Wide-coverage semantic analysis with Boxer. In *Proceedings of the STEP 2008 Conference*, pages 277–286. College Publications, 2008.

[3] Ted Briscoe. An introduction to tag sequence grammars and the RASP system parser. Technical Report UCAM-CL-TR-662, University of Cambridge Computer Laboratory, 2006.

[4] Ted Briscoe and John Carroll. Evaluating the accuracy of an unlexicalized statistical parser on the PARC DepBank. In *Proceedings of the Poster Session of the Joint Conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics (COLING/ACL-06)*, pages 41–48, Sydney, Australia, 2006.

[5] Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Meeting of the ACL*, pages 173–180, Michigan, Ann Arbor, 2005.

[6] Stephen Clark and James R. Curran. Partial training for a lexicalized-grammar parser. In *Proceedings of the Human Language Technology Conference and the Annual Meeting of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL'06)*, pages 144–151, New York, 2006.

[7] Stephen Clark and James R. Curran. Perceptron training for a wide-coverage lexicalized-grammar parser. In *Proceedings of the ACL-07 Workshop on Deep Linguistic Processing*, pages 9–16, Prague, Czech Republic, 2007.

[8] Stephen Clark and James R. Curran. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552, 2007.

[9] Stephen Clark and Julia Hockenmaier. Evaluating a wide-coverage CCG parser. In *Proc. of the LREC 2002 Beyond Parseval Workshop*, pages 60–66, Las Palmas, Spain, 2002.

[10] Michael Collins. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Meeting of the ACL*, pages 16–23, Madrid, Spain, 1997.

[11] Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of EMNLP*, pages 1–8, Philadelphia, USA, 2002.

[12] Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Meeting of the ACL*, pages 111–118, Barcelona, Spain, 2004.

[13] James R. Curran, Stephen Clark, and Johan Bos. Linguistically motivated large-scale NLP with C&C and Boxer. In *Proceedings of the ACL 2007 Demonstrations*, pages 33–36, Prague, Czech Republic, 2007.

[14] James R. Curran, Stephen Clark, and David Vadas. Multi-tagging for lexicalized-grammar parsing. In *Proceedings of the Joint Conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics (COLING/ACL-06)*, pages 697–704, Sydney, Australia, 2006.

[15] Darren Foong. Optimising a natural language parser in Java. Part II dissertation, University of Cambridge, 2015.

[16] Julia Hockenmaier and Mark Steedman. Generative models for statistical parsing with Combinatory Categorial Grammar. In *Proceedings of the 40th Meeting of the ACL*, pages 335–342, Philadelphia, PA, 2002.

[17] Julia Hockenmaier and Mark Steedman. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396, 2007.

[18] Liang Huang, Suphan Fayong, and Yang Guo. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151, Montreal, Canada, 2012.

[19] P. Liang, A. Bouchard-Cote, D. Klein, and B. Taskar. An end-to-end discriminative approach to machine translation. In *Proceedings of COLING/ACL*, 2006.

[20] Adwait Ratnaparkhi. A maximum entropy part-of-speech tagger. In *Proceedings of the EMNLP Conference*, pages 133–142, Philadelphia, PA, 1996.

[21] Stefan Riezler, Tracy H. King, Ronald M. Kaplan, Richard Crouch, John T. Maxwell III, and Mark Johnson. Parsing the Wall Street Journal using a Lexical-Functional Grammar and discriminative estimation techniques. In *Proceedings of the 40th Meeting of the ACL*, pages 271–278, Philadelphia, PA, 2002.

[22] Laura Rimell and Stephen Clark. Adapting a lexicalized-grammar parser to contrasting domains. In *Proceedings of the 2008 EMNLP Conference*, pages 475–484, Honolulu, Hawai‘i, 2008.

[23] Nathan Schneider, Brendan O’Connor, Naomi Saphra, David Bamman, Manaal Faruqui, Noah A. Smith, Chris Dyer, and Jason Baldridge. A framework for (under)specifying dependency syntax without overloading annotators. In *Proc. of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, Sofia, Bulgaria, 2013.

[24] Mark Steedman. *The Syntactic Process.* The MIT Press, Cambridge, MA, 2000.

[25] Wenduan Xu, Michael Auli, and Stephen Clark. CCG supertagging with a recurrent neural network. In *Proceedings of the short papers of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL 2015)*, Beijing, China, 2015.

[26] Wenduan Xu, Stephen Clark, and Yue Zhang. Shift-reduce CCG parsing with a dependency model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL 2014)*, pages 218–227, Baltimore, MD, 2014.

[27] H. Zhang, L.Huang, K. Zhao, and R. McDonald. Online learning for inexact hypergraph search. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP)*, Seattle, WA, 2013.

[28] H. Zhang and R. McDonald. Generalized higher-order dependency parsing with cube pruning. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP)*, Jeju, Korea, 2012.

[29] Yue Zhang and Stephen Clark. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP-08)*, pages 562–571, Honolulu, Hawai‘i, 2008.

[30] Yue Zhang and Stephen Clark. Shift-reduce CCG parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: HLT (ACL 2011)*, pages 683–692, Portland, OR, 2011.