Rapid Incremental Parsing with Repair

Steven P. Abney

1990

Abstract

This work describes a method of achieving rapid, reliable parsing of natural text through the application of three techniques: (1) resolving small questions sequentially, (2) repairing errors directly, instead of searching through a non-deterministic space, and (3) recognizing major constituents before analyzing the details of their internal structure. The resulting parser, which I call CASS, is fast and accurate. It parses a million words in 5-6 hours; that is as fast as the fastest parsers reported in the literature. Its accuracy at recognizing chunks (mid-level constituents) and at identifying subjects and predicates is 95% or better.

Introduction

Much of the information contained in online texts will remain inaccessible until we have natural language parsers that are faster and more reliable than the current state of the art. In the work described here I explore the possibility of achieving rapid, reliable parsing of natural text by application of some simple techniques: (1) resolving small questions sequentially, (2) repairing errors from earlier processing as they become apparent downstream, and (3) recognizing major constituents before parsing them, in order to assure large-scale well-formedness before expending time on detailed analysis.

I have implemented this approach in a parser called CASS (Cascaded Analysis of Syntactic Structure). It is a pipeline of simple filters. It is 'semi-deterministic', in the sense that each filter makes a definite decision about a specific problem, such as part-of-speech disambiguation or identifying the subject and predicate of simplex clauses. There may be search within filters, but ambiguities are not propagated downstream. When errors become apparent downstream, the parser attempts to repair them. In contrast to backtracking, in which a parser unwinds its computation to an earlier state and pursues a different branch of a nondeterministic computation, *repair* consists in directly modifying erroneous structure without regard to the history of computation that produced the structure.

Deterministic parsers, such as Marcus' PARSIFAL [Marcus 1980] or Hindle's FIDDITCH [Hindle 1983], are generally very fast. For example, FIDDITCH can

parse the megaword Brown corpus in five to six hours [Hindle, p.c.]. Unfortunately, they do not scale well. FIDDITCH is the only Marcus-style parser I know of with a realistically large grammar. However, it represents a considerable investment in human labor, and probably is close to the limit of what is possible for this type of parser. Mainstream parsers, such as chart parsers, explore the space of possible parses more systematically. As a result, the grammar writer is freed (in principle) from details of processing, making the construction and management of much larger grammars feasible. The downside is that the increased amount of search substantially slows parsing. By factoring the parsing problem into a sequence of small, well-defined questions, CASS addresses the problem of scalability without sacrificing the speed of deterministic parsing.

An objection to tackling ambiguities sequentially is that there are generally some low-level ambiguities (e.g., part-of-speech ambiguities) that cannot be definitively resolved without reference to higher-level syntactic context. If we use only low-level context, we cannot achieve error-free parsing. But, of course, no parser correctly parses every sentence occuring in natural text. The salient performance question is not errors versus no errors, but the tradeoff between speed and error rate. Moreover, recent work in stochastic part-of-speech tagging [Church 1988, DeRose 1988, Garside 1987] shows that it is possible to resolve part-of-speech ambiguity with considerable reliability, considering only local word-level context. We can accept the remaining errors as the price of reducing the amount of search required, and attempt to repair them when they are detected in the course of higher-level processing.

1 The Parser

CASS takes as its input the output of Church's POS (Part Of Speech) program [Church 1988]. POS tags words with their part of speech, and it also recognizes non-recursive NP's. (A non-recursive NP is the segment of an NP from the first word to the head noun.) CASS consists of three main filters:

1. The **Chunk Filter** builds chunks. It corrects some common errors made by POS's NP-recognition component.

2. The **Clause Filter** recognizes clauses. It identifies the beginning and end of simplex clauses, and marks the subject and predicate. If it does not find a unique subject and predicate, it attempts error correction.

3. The **Parse Filter** assembles chunks into complete parse trees. Its primary tasks are dealing with conjunction and attachment.

These filters decompose further into subfilters performing even simpler transformations of the parse stream.

1.1 POS

POS tags words with their parts of speech, and marks non-recursive NP's. I have added a postprocessor to translate the POS parts of speech to more familiar categories, and to mark some special words and collocations (*only, ago, more than, such a*, etc.)

The following is an example of the output (text from [Williams 1986]).

> CS In_{p} [$\text{South}_{\text{pnoun}}$ $\text{Australia}_{\text{pnoun}}$ beds_{npl}] of_{p} [$\text{boulders}_{\text{npl}}$] were_{bed} $\text{deposited}_{\text{vbn}}$ by_{p} [$\text{melting}_{\text{vbg}}$ $\text{icebergs}_{\text{npl}}$] in_{p} [a_{det} gulf_{n}] [that_{wps}] $\text{marked}_{\text{vbd}}$ [the_{det} $\text{position}_{\text{n}}$] of_{p} [the_{det} $\text{Adelaide}_{\text{pnoun}}$ $\text{geosyncline}_{\text{n}}$] , [an_{det} $\text{elongated}_{\text{adj}}$] , [$\text{sediment-filled}_{\text{adj}}$ $\text{depression}_{\text{n}}$] in_{p} [the_{det} crust_{n}] . CS

(Part-of-speech codes that may be unfamiliar: CS = Clause Separator; Bed = past tense of be; Vbd = past tense verb; Vbg = present participle; Vbn = past participle; Wps = subject wh-pronoun.)

POS's NP-recognition component is less reliable than the part-of-speech component, and there are two examples here of the errors it commonly makes. The first noun phrase is run on, and should be split into *South Australia* and *beds*. The next-to-last noun phrase, *an elongated, sediment-filled depression,* is fragmented.

1.2 Chunk Filter

The chunk filter consists of two subfilters: the NP filter and the chunk filter proper.

1.2.1 NP Filter

The NP filter uses a regular expression recognizer to assemble noun phrases based on the brackets provided by the Church NP-recognizer. It corrects the most common errors involving fragmented NP's, such as fragmentation resulting from conjunction of prenominal adjectives or modification of adjectives/cardinals by adverbs and qualifiers. It also corrects clear cases of run-on NP's, such as when a Det or Pron occurs in non-initial position.

1.2.2 Chunk Filter Proper

The chunk filter proper also uses a regular expression recognizer, with the following grammar (glossing over details):

PP → (P | To)+ (NP | Vbg)
WhPP → (P | To)+ WhNP
AdvP → (QL | Adv)* Adv
AP → (AdvP | QL)* Adj
VP → AdvP? ((MD VP-inf?)
| V-tns
| (Hv-tns VP-perf?)
| (Be-tns VP-prog?)
| (Be-tns VP-pass?))
Inf → To AdvP? VP-inf

(I have omitted the definitions of VP-inf, etc., for brevity's sake. To = infinitival to; QL = qualifier; MD = modal.)

Here is the same example sentence as above, after it has passed through the chunk filter:

CS
[PP In [NP South Australia beds]]
[PP of [NP boulders]]
[VP were deposited]
[PP by [NP melting icebergs]]
[PP in [NP a gulf]]
[WhNP that]
[VP marked]
[NP the position]
[PP of [NP the Adelaide geosyncline]]
,
[NP an elongated, sediment-filled depression]
[PP in [NP the crust]]
.
CS

The next-to-last NP has been repaired, but the parser has not yet recognized that there is an error in the first NP.

1.3 Clause Filter

The clause filter consists of two subfilters: the raw clause filter and the repaired clause filter.

1.3.1 Raw Clause Filter

The raw clause filter attempts to identify the beginning and end of each simplex clause, and mark the subject and predicate. If no unique subject and predicate can be identified, it identifies the type of error encountered—too many VP's, no subject, etc. Again, glossing over details:

Extra-VPs → CM ... NP ... VP ... (VP ...)+
Clause → CM ... NP ... VP ...
ORC → CM? WhNP ... NP ... VP ...
SRC → CM? WhNP ... VP ...
WhClause → CM? (WhPP | Whadv) ... NP ... VP ...
VP-conj → Conj VP ...
No-subj → CM ... VP ...
No-VP → CM ...

(CM = "Clause Marker", i.e., elements like complementizers, conjunctions, periods, etc. that potentially mark a clause division; SRC = subject relative clause; ORC = object relative clause.)

1.3.2 Clause Repair Filter

The parser attempts to repair No-subj and Extra-VPs clauses. (No-VP phrases are usually not genuine errors. They are merged into the previous clause.) The clause repair filter recognizes three kinds of errors that result in No-subj clauses:

1. Unrecognized Partitives

   $(\text{much} \mid \text{many} \mid \text{most})_{\text{adv}}$ $[_{\text{PP}}$ of ...$]$ → $[_{\text{NP}}$ $(\text{much} \mid \text{many} \mid \text{most})_{\text{n}}]$ $[_{\text{PP}}$ of ...$]$

2. Run-on NP

   Split between suspicious pairs of categories: PNOUN NPL, NPL ADJ, etc.

   E.g. $[_{\text{PP}}$ In $[_{\text{NP}}$ $\text{South}_{\text{pnoun}}$ $\text{Australia}_{\text{pnoun}}$ $\text{beds}_{\text{npl}}]]$ → $[_{\text{PP}}$ In $[_{\text{NP}}$ $\text{South}_{\text{pnoun}}$ $\text{Australia}_{\text{pnoun}}]$ $[_{\text{NP}}$ $\text{beds}_{\text{npl}}]]$

3. Misanalyzed Complementizer

   $[_{\text{PP}}$ $\text{x}_{\text{p-time}}$ NP] → x_{c} NP

   E.g. $[_{\text{PP}}$ Before_{p} $[_{\text{NP}}$ winter$]]$ $[_{\text{VP}}$ arrived$]$ → Before_{c} $[_{\text{NP}}$ winter$]$ $[_{\text{VP}}$ arrived$]$

There are three common cases for Extra-VPs clauses, as well.

1. Null Relative Pronoun

   (NP | PP) ... NP ... VP → (NP | PP) ... $[_{\text{ORC}}$ NP ... VP]

   E.g. $[_{\text{PP}}$ in the experiment$]$ $[_{\text{ORC}}$ $[_{\text{NP}}$ I$]$ $[_{\text{VP}}$ described$]]$

2. Null Complementizer

   VP-scomp ... NP ... VP → VP-scomp ... $[_{\text{Clause}}$ NP ... VP]

   (VP-scomp = VP whose head is subcategorized for a sentential complement)

   E.g. $[_{\text{VP}}$ said$]$ $[_{\text{Clause}}$ $[_{\text{NP}}$ the test$]$ $[_{\text{VP}}$ was$]]$

3. Misanalyzed Complementizer

   $[_{\text{PP}}\ x_{\text{p-time}}\ \text{NP}\ ]\ \ldots\ \text{VP} \rightarrow [_{\text{Clause}}\ x_{\text{c}}\ \text{NP}\ \ldots\ \text{VP}\ ]$

If the clause repair filter cannot match one of these specific templates, it guesses that any NP ... VP pair constitutes a clause, and any VP standing alone is a predicate. Finally, any complementizer (other than "that") that does not turn out to introduce a clause is taken to be a misanalyzed preposition. E.g.

$\text{than}_{\text{c}}\ [_{\text{NP}}\ \text{John}] \rightarrow [_{\text{PP}}\ \text{than}_{\text{p}}\ [_{\text{NP}}\ \text{John}]]$

The following is an example of the output of the clause filter:

Begin-No-subj
[PP In South Australia]
[Subj [NP beds]]
[PP of boulders]]
[Pred [VP were deposited]]
[PP by melting icebergs]]
[PP in a gulf]]
Begin-SRC
[Subj [WhNP that]]
[Pred [VP marked]]
[NP the position]
[PP of the Adelaide geosyncline]
,
[NP an elongated, sediment-filled depression]
[PP in the crust]

At the clause level, there is finally sufficient information to repair the error in the first NP.

1.4 Parse Filter

Unlike the earlier filters, the parse filter is not essentially a regular-expression recognizer. It assembles recursive structures by attaching nodes one to another, in accordance with the ability of head words to license phrases of various categories as arguments or modifiers. A chunk Y may be attached to chunk X only if the head of X can take Y as argument or modifier. At each point, the parser considers attaching the current chunk either to the immediately preceding chunk, or to the most recent preceding verbal chunk.

There are also a handful of regular expression templates for those cases that do not naturally fall under the licensing paradigm. These fall into two classes: templates that essentially parse what the clause filter recognized (e.g. *S → Begin-Clause Subj Pred*), and templates for assembling conjuncts, e.g. *NP-conj → (Comma NP)* Comma? Conj NP*. When the conjunct has been assembled, it

is attached in the same manner as an argument or modifier. Note that categories mentioned in a template need not be adjacent in the input stream. For example, after the Subj is encountered, arbitrarily many chunks may be attached to the Subj or to subsequent chunks, before the Pred is finally encountered.

The parser chooses one of the possible next actions on the basis of some simple heuristics. If there is no applicable action, the parser "punts" the current chunk, marking its general location in the tree, but leaving the exact attachment site indeterminate.

2 Evaluation

Researchers generally recognize the importance of evaluating parser performance, but there is little discussion of evaluation techniques in the literature. In the absence of a standard evaluation protocol, I have applied some fairly simple tests to CASS to gauge its speed and accuracy, as described in this section.

I parsed two samples from a corpus of 180 Scientific American articles. The first, the training sample, was a single article that I used during development of the parser. The second, the test sample, was a random sample of sentences from the remaining 179 articles.

2.1 Speed

I was primarily concerned with performance along two dimensions: speed and accuracy. The following are data on parsing speed on a Sun 4.

TRAINING:	65.2 sec (averaged over 20 runs)
size:	27407 bytes, 4378 words, 176 sentences
rate:	67.1 w/s (words per second)
	4:08 h/MW (hours per megaword)
TEST:	27.5 sec (averaged over 70 runs)
size:	10202 bytes, 1663 words, 78 sentences
rate:	60.4 w/s
	4:36 h/MW

Parse times are exclusive of time spent in the POS program, which runs at 239 w/s = 1:10 h/MW on a Sun 4. Hence, times for POS + CASS would be 5:18 or 5:46 h/MW.

These times are most useful as an indicator of whether this processing is practical for moderate to large corpora. The Brown and LOB corpora are 1 MW each; they could be processed in five or six hours. The combined A, B, and F wires of the Associated Press produce somewhat more than a third of a megaword of raw text per day; parsing their output would require a little under

two hours per day. Of course, what is practical varies greatly with hardware. On a Symbolics UX400S, CASS runs substantially slower, at 7:37 h/MW vs. 4:08-4:36 h/MW. (On the Symbolics, parsing rates for training and test samples are virtually identical. I do not know why they vary so much on the Sun.)

The times for POS + CASS are as good as the fastest parse times reported in the literature. FIDDITCH does somewhat more analysis than CASS, and parses a megaword in five or six hours, i.e., the same as POS + CASS. CLARIT [Evans 1989] appears to do somewhat less analysis than CASS, and has a reported speed of 600 words/min = 28 h/MW. McDonald's parser [McDonald 1990] runs at 8–20 w/s (14–35 h/MW) on a Macintosh II, depending on completeness of parse. Of course, such comparisons are suggestive at best, because of numerous uncontrolled variables.

2.2 Accuracy

Accuracy is rather more difficult to measure than speed. I have limited myself to asking two fairly well-defined questions:

(1) How many errors did the parser make in deciding (a) where each chunk begins and ends, and (b) the category of each chunk? I did not count punctuation, conjunctions, or complementizers as chunks. I collected two types of statistics: actual chunks that were erroneously categorized or segmented, and observed chunks (those output by the parser) that were incorrect with regard to categorization or segmentation. The two measures are correlated, but not identical: run-on chunks get counted as multiple errors under the first measure, but not the second, whereas fragmented chunks get counted as multiple errors under the second measure, but not the first.

(2) How many errors did the parser make in identifying subjects and predicates? These counts are based on actual chunks, not observed chunks. I include separate statistics for false positives and false negatives. I also give the total probability of a chunk being mislabelled.

Even though these are fairly concrete questions, a good deal of judgment is required in answering them, so allowance must be made for subjectivity, when interpreting the following data.

	TRAINING		TEST	
Actual chunks:	1797		732	
Erroneous:	78	= 4.34%	36	= 4.92%
Observed chunks:	1821		739	
Erroneous:	102	= 5.60%	44	= 5.95%
Subjs & preds:	625		295	
False neg:	53	= 8.48%	19	= 6.44%
Non-subj/pred:	1172		437	
False pos:	26	= 2.22%	14	= 3.20%
Total (= actual chunks):	1797		732	
Erroneous:	79	= 4.40%	33	= 4.51%

Two notes: the errors on subjects and predicates include errors caused by erroneously categorized/segmented chunks. If we only consider good chunks, the error rate for e.g. false negatives drops from 8.48% training / 6.44% test to 4.46% training / 3.96% test. Second, perhaps the most informative aspect of the data is that performance remains about the same when moving from the training to the test sample, indicating that the parser is tuned to properties of the text that are valid generally, and not just for the text used in parser development.

Conclusion

CASS provides an initial validation of the principles it embodies, namely, sequentially addressing small, well-defined questions, repairing errors as they become apparent in downstream processing, and recognizing large-scale syntactic features before doing detailed analysis. CASS is extremely fast and reliable. It parses a megaword in under six hours. It is about 95% accurate in categorizing and segmenting chunks, and it is about 95% accurate in labelling chunks as subject, predicate, or neither. All in all, this line of attack appears to hold out great promise for rapid, reliable parsing of unrestricted text.

Bibliography

Church, Kenneth (1988) "A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text," *Proceedings of the Second ACL Conference on Applied Natural Language Processing.*

DeRose, S.J. (1988) "Grammatical Category Disambiguation by Statistical Optimization," *Computational Linguistics* 14(1):31-39.

Hindle, Donald (1983) "User manual for Fidditch," Naval Research Laboratory Technical Memorandum #7590-142.

Marcus, Mitchell (1980) *A Theory of Syntactic Recognition for Natural Language,* The MIT Press, Cambridge, Massachusetts.

McDonald, David D. (1990) "Robust Partial-Parsing through Incremental, Multi-level Processing: Rationales and Biases," in *Text-Based Intelligent Systems,* Working Notes, AAAI Spring Symposium, Stanford University.

Garside, Roger (1987) "The CLAWS word-tagging system," in Garside et al., eds., *The Computational Analysis of English,* Longman, New York.

Williams, George E. (1986) "The Solar Cycle in Precambrian Time," *Scientific American* 255(2): 88-97.