EFFICIENCY IN LARGE-SCALE PARSING SYSTEMS

PROCEEDINGS

a workshop held at Coling 2000
the 18th International Conference on Computational Linguistics
Luxembourg, 5 August 2000

Organisation

Organisers

John Carroll

Robert C. Moore

Stephan Oepen

Programme Committee

To obtain additional copies of these proceedings contact:

Dr. John Carroll
Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH, UK

`john.carroll@cogs.susx.ac.uk`

Contents

Invited Talk

Why not Cubic?

Ronald M. Kaplan
Xerox PARC
3333 Coyote Hill Road, Palo Alto
CA 94304, USA
kaplan@parc.xerox.com

It is well-established that the parsing problem for higher-level constraint-based formalisms such as LFG, PATR, and HPSG is in the NP-hard complexity class. Thus there are worst-case sentences and grammars for which there are no known polynomial algorithms. Unhappily, the most straightforward parsers for these formalisms tend to be exponential not only in the worst cases but also for the common cases of sentences and grammars that intuitively seem to be less complex. Research aimed at improving performance has typically accepted the inherently exponential nature of the problem and has then focused on implementation techniques that can lower the space/time computational resource curves but without actually changing their shape.

In this talk I will discuss an alternative stragegy that we have been exploring in our work on LFG parsing. Instead of taking the exponential as a given for arbitrary grammars and asking how we can make it more palatable, we studied a restricted class of LFG grammars whose languages are obviously context-free. We then asked a different question: why doesn't a conventional LFG parser recognize these languages in cubic time, their theoretically obtainable bound? We developed a few key ideas that taken together provide for cubic performance for the special case of a completely context-free-equivalent LFG grammar, and provide nearly cubic performance for the less restricted set of LFG grammars that are linguistically relevant.

Papers

Efficient Large-Scale Parsing — a Survey

John Carroll
Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH, UK
johnca@cogs.susx.ac.uk

Stephan Oepen
Computational Linguistics
Saarland University
66041 Saarbrücken, Germany
oe@coli.uni-sb.de

Abstract

We survey work on the empirical assessment and comparison of the efficiency of large-scale parsing systems. We focus on (1) grammars and data used to assess parser efficiency; (2) methods and tools for empirical assessment of parser efficiency; and (3) comparisons of the efficiency of different large-scale parsing systems.

1 Background

Interest in large-scale, grammar-based parsing has recently seen a large increase, in response to the complexities of language-based application tasks such as speech-to-speech translation, and enabled by the availability of more powerful computational resources, and by efforts in large-scale and collaborative grammar engineering and also in the induction of statistical grammars/parsers from treebanks.

There are two main paradigms in the evaluation and comparison of the performance of parsing algorithms and implemented systems: (i) the formal, complexity-theoretic analysis of how an algorithm behaves, typically focussing on worst-case time and space complexity bounds; and (ii) the empirical study of how properties of the parser and input (possibly including the grammar used) affect actual, observed run-time efficiency.

It has been shown (Maxwell and Kaplan, 1993; Carroll, 1994; van Noord, 1997) that the theoretical study of algorithms alone does not (yet) suffice to provide an accurate prediction about how a specific algorithm will perform in practice, when used in conjunction with a specific grammar (or type of grammar), and when applied to a particular domain and task. Therefore, empirical assessment of practical parser performance has become an established technique and continues to be the primary means of comparison among algorithms. At the same time, system competence (i.e. coverage and overgeneration with respect to a particular grammar and test set) cannot be decoupled from the evaluation of parser performance, because two algorithms can only be compared meaningfully when they really solve the same problem. This typically means that they either directly use the same grammar, or at least achieve demonstrably similar competence on the same test set.

In the next section, we briefly describe large-scale grammars and test suites that have been used in evaluations of parser efficiency. Section 3 discusses methods and computational tools that have been used in such evaluations, and Section 4 surveys research comparing the efficiency of different parsers or parsing strategies with large-scale grammars.

2 Grammars and Data

A number of large-scale, general-purpose grammars have been used in evaluations of parser efficiency. We describe their main characteristics briefly below.[1]

- The Alvey NL Tools (ANLT) contains a large, wide-coverage sentence grammar of English (Grover, Carroll, & Briscoe, 1993), written in a unification-based metagrammatical formalism resembling GPSG. The grammar expands out to an object grammar of 780 DCG-

[1]While it is the case that most current large-scale grammar-based parsing systems construct constituent structure representations that are capable of supporting semantic interpretation, the English Constraint Grammar (Karlsson, Voutilainen, Heikkilä, & Anttila, 1995) and the Link Grammar (Sleator & Temperley, 1993) systems are exceptions. Thus, since the motivations behind these grammars are different we do not consider them here.

like rules, each category containing on average around 30 nodes. Associated with the grammar is a test suite, originally written by the grammarian to monitor coverage during grammar development, containing around 1,400 (mostly grammatical) items.

- The SRI Core Language Engine (CLE) grammar (Alshawi, 1992) is also GPSG-inspired, but with different treatments of a number of central syntactic phenomena, such as subcategorisation and unbounded dependencies. The grammar contains of the order of 150 rules which map fairly directly into a DCG.
- The LinGO English grammar (Flickinger & Sag, 1998) is a broad-coverage HPSG developed at CSLI Stanford. The grammar contains roughly 8,000 types and 64 lexical and grammar rules, with an average feature structure size of around 300 nodes. Three main test sets have been used for parser evaluation with this grammar, the largest—containing 2,100 items—having been extracted from VerbMobil corpora of transcribed speech and balanced with respect to sentence length. (Comparable grammars of German and Japanese, again originally developed in VerbMobil, are shortly to be available).
- In the Xerox-led ParGram collaboration (Butt, King, Niño, & Segond, 1999), wide-coverage grammars of English, French, German and a number of other languages are being developed in parallel in the LFG framework, all of the grammars based on a common set of linguistic principles, with a commonly-agreed-upon set of grammatical features. Each grammar consists of an atomic-categoried phrase-structure backbone augmented with feature annotations.
- The trees in the Penn Treebank induce a large context-free grammar containing 15,000 rules. A recent comparison of context-free parsing strategies (Moore, 2000) has used this grammar, a second one derived from an ATIS treebank (with 4,600 productions), and a third (24,500 productions) produced by computing an atomic-categoried backbone from a unification-based phase structure grammar. Test sentences for these grammars were derived either from the associated corpora, or artificially, by using the grammar to stochastically generate random strings.
- The XTAG system grammar (XTAG, 1995) is a large-scale lexicalised tree adjoining grammar of English, developed by several researchers over the past ten years or so. The grammar contains of the order of 500 elementary tree schemata, organised into families; each lexeme is associated with a number of these families. Nodes in the tree schemata are augmented with feature structures so that information can be passed non-locally between elementary trees.

Test suites supplied with grammars have typically been written by the grammar developers themselves for the purpose of monitoring over- and under-generation as the grammar is changed. However, the test suites have also been found to be of some value for evaluating parser efficiency. A major drawback in this context, though, is that each test suite item usually only contains very limited ambiguity (easing the task of checking the resulting parses), and is relatively short (so that only one or two constructions are tested at a time). This is also the case for independently-developed test suites, such as the TSNLP suites for English, French and German (Oepen, Netter, & Klein, 1997). Therefore, in some parser evaluation work, new suites of longer sentences have had to be constructed manually or extracted specially from corpora.

Another important issue is the degree to which the grammars are available to the general NL processing research community. Those developed within companies are in general more difficult to obtain, although use for parser evaluation may be easier to negotiate than use within an actual application system, for instance.

3 Methods and Tools

Previous work on the assessment and comparison of large-scale parsers has mostly been concerned with evaluation of parser (or grammatical) coverage, and with correctness of the analyses produced. So, for example, coverage has been expressed in terms of lists of grammatical phenomena for which an analysis is provided; over- and under-generation as the percentage of grammatical or ungrammatical items from a given reference set that are or are not assigned some sort of analysis; and degree of ambiguity of a grammar in terms of the 'parse base', the expected number of parses for a given input length (Carroll, Briscoe, & Sanfilippo, 1998). Work on quantifying parse correctness has used various measures of structural consistency with respect to constituent structure annotations of a corpus (e.g. exact match, crossing brackets, tree similarity, and others—see Black et al., 1991, Black, Garside, & Leech, 1993, Grisham, Macleod, & Sterling, 1992, and Briscoe & Carroll, 1993); recently, more general schemes have been advocated that deploy functor – argument (dependency) relations as an abstraction over different phrase structure analyses that a parser may assign (Lin, 1995; Lehmann et al., 1996; Carroll et al., 1998). The Penn Treebank and the SUSANNE corpus are well-established resources for the evaluation of parser accuracy.

In a sharp contrast, there is little existing methodology, let alone established reference data or software tools, for the evaluation and contrastive comparison of parser efficiency. Although most grammar development environments and large-scale parsing systems supply facilities to batch-process a test corpus and record the results produced by the system, these are typically restricted to processing a flat, unstructured input file (listing test sentences, one per line) and outputting a small number of processing results to a log file.[2] Additionally, no metrics exist that allow the comparison of parser efficiency across different grammars and sets of reference data. We therefore note a striking methodological and technological deficit in the area of precise and systematic assessment of grammar and parser behaviour.

Recently though, a new methodology, termed *competence & performance profiling* (Oepen & Flickinger, 1998; Oepen & Carroll, 2000), has been proposed that aims to fill this gap. Profiles are rich, precise, and structured snapshots of parser competence (coverage and correctness) and performance (efficiency), where the production, maintenance, and inspection of profiles is supported by a specialised software package called [incr tsdb()].[3] Profiles are stored in a relational database that serves as the basis for flexible report generation, visualisation, data analysis via basic descriptive statistics, and of course comparison to other profiles. The [incr tsdb()] package has so far been interfaced with some eight unification-based grammar development and/or parsing systems, and has served as the 'clearing house' in a multi-site collaborative effort on parser benchmarking (Flickinger, Oepen, Tsujii, & Uszkoreit, 2000), resulting in useful feedback to all participating groups.

4 Efficiency Comparisons

Many parsing algorithms suitable for NL grammars have been proposed over the years, their proponents often arguing that the number of computational steps are minimised with respect to alternative, competing algorithms. However, such arguments can only be made in the case of very closely related algorithms; qualitatively different computations can only reliably be compared empirically. So, for example, generalised LR parsing was put forward as an improvement over Earley-style parsing (Tomita, 1987), with a justification made by running implementations of the two types of parser on a medium-sized CF grammar with attribute-value augmentations. However, comparisons of this type have to be done with care. The coding of different strategies must use exactly equivalent techniques, and to be able to make any general claims, the grammar(s) used must be large enough to fully stress the algorithms. In particular, with grammars admitting less ambiguity, parse time is likely to increase more slowly with increasing input length, and also with smaller grammars rule application can be constrained tightly with relatively simple predictive techniques. In fact, a more recent evaluation (Moore, 2000) using a number of large-scale CF grammars has shown conclusively that generalised LR parsing is less efficient than certain left-corner parsing strate-

[2] Some (Meta-)Systems like PLEUK (Calder, 1993) and HDrug (van Noord & Bouma, 1997) that facilitate the exploration of multiple descriptive formalisms and processing strategies come with slightly more sophisticated benchmarking facilities and visualisation tools. However, they still largely operate on monolithic, unannotated input data sets, restrict accounting of system results to a small number of parameters (e.g. number of analyses, overall processing time, memory consumption, possibly the total number of chart edges), and only offer a limited, predefined choice of analysis techniques.

[3] See '`http://www.coli.uni-sb.de/itsdb/`' for the (draft) [incr tsdb()] user manual, pronunciation guidelines, and instructions on obtaining and installing the package.

gies.

Moore and Dowding (1991) document a process of refining a unification-based (purely bottom-up) CKY parser (forming part of a speech understanding system) by incorporating top-down information to prevent it hypothesising constituents bottom-up that could not form part of a complete analysis, given the portions of rules already partially instantiated. An important step was reducing the spurious prediction of gaps by means of grammar transformations. The refinement process was guided throughout by empirical measurements of parser throughput on a test corpus.

Improvements in efficiency can be gained by specialising a general-purpose grammar to a particular corpus. Samuelsson and Rayner (1991) describe a machine learning technique that is applied to the CLE grammar to produce a version of the grammar that parses ATIS corpus sentences much faster than the original grammar. In general there are more rules in the specialised grammar than in the original, but they are more specific and can thus be applied more efficiently.

Maxwell and Kaplan (1993) investigate the interaction between parsing with the CF backbone component of a grammar and the resolution of functional constraints, using a precursor of the English ParGram grammar. A number of parsing strategies are evaluated, in combination with two different unifiers, on a small set of test sentences. There is a wide gap between the best and worst performing technique; the differences can be justified intuitively, but not with any formal analyses of computational complexity.

Carroll (1994) discusses the throughput of three quite distinct unification-based parsing algorithms running with the ANLT grammar. The main findings were that exponential parsing algorithm complexities with respect to grammar size have little impact on the performance of the parsers, since they all achieved relatively good throughput, and parse table sizes were also quite manageable. Increases in parse times with longer inputs were also fairly controlled, being roughly only quadratic. In another experiment, running the ANLT grammar with the CLE parser resulted in very poor performance, suggesting that the parallel development of the software and grammars had inadvertently caused them to become 'tuned' to one another.

van Noord (1997) presents an efficient implementation of head-corner parsing, as used in a prototype spoken language dialogue system. Memoisation and goal-weakening techniques are used to reduce parser space requirements; the head-corner parser also runs faster than implementations of left-corner, bottom-up and LR parsers in evaluations using a DCG of Dutch with speech recogniser word-graph input. A further set of evaluations use the ANLT grammar, allowing a tentative cross-system comparison with the ANLT parser to be made.

In work concerned with parsing with large-scale CF grammars, Moore (2000) investigates empirically the interactions between various types of grammar factoring and versions of the left-corner parsing algorithm that differ in the details of precisely how and in what order top-down filtering information is applied. Using three very different grammars, one of the parser/factoring combinations was found to be consistently and significantly better than the alternatives, despite being only minimally different from the other variants. This strategy was also shown to outperform several other major approaches to CF parsing.

Sarkar (2000) evaluates the efficiency of a chart-based head-corner parsing algorithm on a corpus of 2,250 Wall Street Journal sentences, using a large-scale grammar (containing 6,800 elementary tree schemata) extracted automatically from the Penn Treebank. For each sentence, parse times were found to correlate roughly exponentially with the number of lexicalised elementary trees selected; there was little correlation between sentence length and parse time.

Oepen and Carroll (2000) describe and argue for a strategy of *performance profiling* in the engineering of parsing systems for wide-coverage linguistic grammars. The aim is to characterise system performance at a very detailed technical level, but at the same time to abstract away from idiosyncracies of particular processing systems. Based on insights gained from detailed performance profiles of various parsing strategies with the LinGO English grammar, a novel 'hyper-active' parsing strategy is synthesised and evaluated.

A number of other empirically-driven research efforts into efficient parsing are described in the same journal special issue (Flickinger et al., 2000). These include grammar-writing techniques for improved parser efficiency, new efficient algorithms for feature structure operations, fast pre-unification filtering, and techniques for the extraction of CF grammars and abstract machine compilation for HPSGs.

5 Conclusions

Recent interest in large-scale, grammar-based parsing (in response to the demands of complex language-based application tasks) has led to renewed efforts to develop wide-coverage, general-purpose grammars, and associated research efforts into efficient parsing with these grammars. Some initial progress has been made towards precise empirical assessment of parser efficiency. However, more work is needed on methods, standard reference grammars and test data to facilitate improved comparability.

Acknowledgements

The first author is supported by a UK EPSRC Advanced Fellowship, and the second by the *Deutsche Forschungsgemeinschaft* Collaborative Research Division *Resource-Adaptive Cognitive Processes* (SFB 378) project B4 (PERFORM).

References

Alshawi, H. (Ed.). (1992). *The Core Language Engine.* Cambridge, MA: MIT Press.

Black, E., Abney, S., Flickenger, D. P., Gdaniec, C., Grishman, R., Harrison, P., Hindle, D., Ingria, R., Jelinek, F., Klavans, J., Liberman, M., Marcus, M., Roukos, S., Santorini, B., & Strzalkowski, T. (1991). A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of the 4th DARPA speech and natural language workshop.* Pacific Grove, CA: Morgan Kaufmann.

Black, E., Garside, R., & Leech, G. (Eds.). (1993). *Statistically-driven computer grammars of English. The IBM – Lancaster approach.* Amsterdam, The Netherlands: Rodopi.

Briscoe, E., & Carroll, J. (1993). Generalised probabilistic LR parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics, 19 (1)*, 25 – 60.

Butt, M., King, T. H., Niño, M.-E., & Segond, F. (1999). *A grammar writer's cookbook.* Stanford, CA: CSLI Publications.

Calder, J. (1993). Graphical interaction with constraint-based grammars. In *Proceedings of the 3rd Pacific Rim Conference on Computational Linguistics* (pp. 160 – 169). Vancouver, BC.

Carroll, J. (1994). Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *Proceedings of the 32nd Meeting of the Association for Computational Linguistics* (pp. 287 – 294). Las Cruces, NM.

Carroll, J., Briscoe, E., & Sanfilippo, A. (1998). Parser evaluation: a survey and a new proposal. In *Proceedings of the 1st International Conference on Language Resources and Evaluation* (pp. 447 – 454). Granada, Spain.

Flickinger, D., Oepen, S., Tsujii, J., & Uszkoreit, H. (Eds.). (2000). *Journal of Natural Language Engineering. Special Issue on Efficient processing with HPSG: Methods, systems, evaluation.* Cambridge, UK: Cambridge University Press. (in press)

Flickinger, D. P., & Sag, I. A. (1998). Linguistic Grammars Online. A multi-purpose broad-coverage computational grammar of English. In *CSLI Bulletin 1999* (pp. 64 – 68). Stanford, CA: CSLI Publications.

Grisham, R., Macleod, C., & Sterling, J. (1992). Evaluating parsing strategies using standardized parse files. In *Proceedings of the 3rd ACL Conference on Applied Natural Language Processing* (pp. 156 – 161). Trento, Italy.

Grover, C., Carroll, J., & Briscoe, E. (1993). *The Alvey Natural Language Tools grammar (4th release)* (Technical Report No. 284). University of Cambridge: Computer Laboratory.

Karlsson, F., Voutilainen, A., Heikkilä, J., & Anttila, A. (1995). *Constraint grammar: a language-independent system for parsing unrestricted text.* Berlin, Germany: Mouton de Gruyter.

Lehmann, S., Oepen, S., Regnier-Prost, S., Netter, K., Lux, V., Klein, J., Falkedal, K., Fouvry, F., Estival, D., Dauphin, E., Compagnion, H., Baur, J., Balkan, L., & Arnold, D. (1996). TSNLP — Test Suites for Natural Language Processing. In *Proceedings of the 16th International Conference on Computational Linguistics* (pp. 711 – 716). Kopenhagen, Denmark.

Lin, D. (1995). A dependency-based method for evaluating broad-coverage parsers. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (pp. 1420 – 1425). Montreal, Canada.

Maxwell III, J. T., & Kaplan, R. M. (1993). The interface between phrasal and functional constraints. *Computational Linguistics, 19 (4)*, 571 – 590.

Moore, R. (2000). Improved left-corner chart parsing for large context-free grammars. In *Proceedings of the 6th International Workshop on Parsing Technologies* (pp. 171–182). Trento, Italy.

Moore, R., & Dowding, J. (1991). Efficient bottom-up parsing. In *DARPA Speech and Natural Language Workshop* (pp. 200–203). Asilomar, CA.

van Noord, G. (1997). An efficient implementation of the head-corner parser. *Computational Linguistics, 23 (3)*, 425 – 456.

van Noord, G., & Bouma, G. (1997). Hdrug. A flexible and extendible development environment for natural language processing. In *Proceedings of the Workshop on Computational Environments for Grammar Development and Linguistic Engineering* (pp. 91 – 98). Madrid, Spain.

Oepen, S., & Carroll, J. (2000). Performance profiling for parser engineering. *Natural Language Engineering, 6 (1) (Special Issue on Efficient Processing with HPSG)*, 81 – 97.

Oepen, S., & Flickinger, D. P. (1998). Towards systematic grammar profiling. Test suite technology ten years after. *Journal of Computer Speech and Language, 12 (4) (Special Issue on Evaluation)*, 411 – 436.

Oepen, S., Netter, K., & Klein, J. (1997). TSNLP — Test Suites for Natural Language Processing. In J. Nerbonne (Ed.), *Linguistic Databases* (pp. 13 – 36). Stanford, CA: CSLI Publications.

Samuelsson, C., & Rayner, M. (1991). Quantitative evaluation of explanation-based learning as an optimization tool for a large-scale natural language system. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence* (pp. 609 – 615). Sydney, Australia.

Sarkar, A. (2000). Practical experiments in parsing using tree adjoining grammars. In *5th international workshop on tree adjoining grammars and related formalisms* (pp. 193 – 198). Paris France.

Sleator, D., & Temperley, D. (1993). Parsing English with a link grammar. In *Proceedings of the 3rd International Workshop on Parsing Technologies* (pp. 277–292). Tilburg, The Netherlands.

Tomita, M. (1987). An efficient augmented-context-free parsing algorithm. *Computational Linguistics, 13 (1)*, 31–46.

XTAG Group (1995). *A lexicalized tree adjoining grammar for english* (Tech. Rep. No. IRCS Report 95-03). The Institute for Research in Cognitive Science, University of Pennsylvania.

Precompilation of HPSG in ALE into a CFG For Fast Parsing

John C. Brown
INVU Services Ltd.,
Blisworth Hill Farm,
Stoke Road,
Blisworth, UK, NN7 3DB
johnbrown@research.softnet.co.uk

Suresh Manandhar
Department of Computer Science,
University of York,
York, UK, YO1 5DD
Suresh@cs.york.ac.uk

Abstract

Context free grammars parse faster than TFS grammars, but have disadvantages. On our test TFS grammar, precompilation into CFG results in a speedup of 16 times for parsing without taking into account additional mechanisms for increasing parsing efficiency. A formal overview is given of precompilation and parsing. Modifications to ALE rules permit a closure over the rules from the lexicon, and analysis leading to a fast treatment of semantic structure. The closure algorithm, and retrieval of full semantic structure are described.

Introduction

Head Driven Phrase Structure Grammar (HPSG), Pollard and Sag (1994) is expressed in Typed Feature Structures (TFSs). Context Free Grammar (CFG) without features supports much faster parsing, but a TFS grammar has many advantages. Fast parsing can be obtained by precompiling a CFG approximation, with TFSs converted into CF near-equivalents. CFG parsing eliminates impossible trees, and TFS unification over the remainder eliminates more, and instantiates path values. Our method treats slashes separately in a precompiled table, and careful allocation of categories to TFSs makes TFS unification unnecessary: instead skeleton semantic structures are formed in parsing, and full structures retrieved afterwards.

A prototype precompiler and fast parser[1] were built in Prolog, and tested with an HPSG grammar of English by Matheson (1996), written in the ALE formalism, by Carpenter and Penn (1996). This has the 6 schemas and 5 principles of HPSG, with 184 lexemes.

A complex sentence, *"kim can believe sandy can expect sandy to persuade kim to promise sandy to try to expect sandy to persuade kim to promise kim to give sandy a happy happy book*", parsed in 3.3s. with retrieval, 5.6 times faster than with TFSs, Brown and Manandhar (2000). An 11 word sentence was 18 times faster at 87ms., or 16 times counting retrieval.

1 Relationship to Work Elsewhere

In precompilation, Torisawa et. al. (2000) repeatedly applied rules leading to maximal projections from lexical heads, allocating categories to mothers and non-head daughters. Our approach allocates categories in a closure over the rules starting with the lexicon, as in Kiefer and Krieger (2000). We differ from both these in that the CFG grammar equates to the TFS grammar, accepting exactly the same strings: a CFG parse tree translates into a TFS tree with no loss of nodes.

We precompiled 550 CF categories and 18,000 rules in 1 hour on a 280MHz. Pentium II. This compares to 5,500 categories and 2,200,000 rules from 11,000 lexemes in 45 hours by Kiefer and Krieger on a 300MHz. Sun Ultrasparc 2, where sets of TFSs in the closure are replaced by common most-general unifiers. The common unifier technique was used to add a CFG back-bone to a unification grammar by Carroll (1993). An `np` has the same *number*, *person* and *gender* features irrespective of an optional specifier and multiple adjectives. A lexical `vp`, partially saturated `vp` and sentence contain decreasing subcategorisation data. Therefore numbers of phrasal and lexical categories are comparable,

[1] Downloadable code on
http://www.cs.york.ac.uk/~johnb or
http://www.soft.net.uk/research/hsppar.htm .

even with exact CF representation. Since this eliminates Kiefer and Krieger's annotation with values of *relevant paths,* large rule numbers are more tolerable.

Our parsing speed-ups are comparable with Kentaro Torisawa's speed-ups of between 47 and 4 in C++ with arrays to store rules and edges, on a 300MHz. Sun Ultrasparc. We used Prolog without constant-time access arrays, with clauses hash-indexed on just the first argument. Global data structures for edges in a chart necessitate dynamic clauses with heavy assertion costs: when referenced, all arguments are treated before matching any against precompiled tables. Tree-structured tables are inefficient, using either one clause per arc with heavy invocation costs, or arguments of nested structures, copied on clause invocation. From full instrumentation revealing bottlenecks, we predict a further speed-up of 10 times, using ideal global data structures in imperative code.

2 Formal View of Precompilation and Parsing

Our CF and TFS grammars are equivalent for two reasons. First, in CF category allocation, TFSs in the list reached by NONLOCAL: INHERITED: SLASH are allocated separate categories and are otherwise ignored. The *filler-head* schema unifies a list member with a TFS from the first daughter, so a precompiled table can predict the outcome during CF parsing. This avoids over-generation of categories formed by alternative slash TFSs in multiple phrasal TFSs arising in long-range dependencies. Second, although semantic sub-structure (also containing word morphology) is omitted in category allocation, its *index* structure may be considered depending on the HEAD value. This makes CFG categories maximally selective. The now unnecessary TFS unification over the CFG parse tree is replaced by semantic TFS retrieval from skeleton structures in constituents.

The method treats grammar rules where mother and daughters each comprise a TFS, expressed as conjoined constraints :

$$TFS_0 \rightarrow TFS_1 \ldots TFS_n \tag{1}$$

where a TFS is given by:

$$\begin{aligned}
&\langle TFS\rangle \rightarrow \langle FS\rangle \mid \langle type\rangle \,\&\, \langle FS\rangle \mid \langle type\rangle\\
&\langle FS\rangle \rightarrow \langle path\rangle : \langle value\rangle \mid \langle FS\rangle \,\&\, \langle FS\rangle\\
&\langle value\rangle \rightarrow \langle TFS\rangle \mid \langle \mathrm{var}\,iable\rangle \,\&\, \langle TFS\rangle\\
&\langle path\rangle \rightarrow \langle feature\rangle \mid \langle feature\rangle : \langle path\rangle
\end{aligned} \tag{2}$$

The value following a *feature* in a TFS of a particular *type* is a sub-type of that given by an appropriateness function $Approp: T \times F \rightarrow T$ where F and T denote finite sets of feature symbols and types. T is organised into a subsumption hierarchy with a single root *bot*.

The grammar may either be lexicalist, or have a large number of specialised rules. Each TFS is *well-formed*, containing no feature not appropriate to its *type*: it is not *totally well-formed* with all appropriate features present. TFS unification involves *type-coercion* which also occurs in constraint application to a TFS, for example of $feature : value$ in a rule. Where a TFS of type t_1 does not contain *feature,* it is coerced into the most general sub-type t_2 of t_1 for which $Approp(t_2, feature)$ is defined.

An integer T_l, corresponding to a CFG category, is allocated in our precompilation to each unique lexical TFS_l, ignoring nested TFSs encoding slashes and semantics, so that

$$\begin{aligned}
T_l = \mathrm{retrieve}^{-1}(&\mathrm{omit}(slash : v1,\\
&\mathrm{re-introduce}(index : v2,\\
&\mathrm{omit}(semantics : v3, TFS_l))))
\end{aligned} \tag{3}$$

where *slash, semantics* and *index* are paths and $v1$, $v2$ and $v3$ are values. Implementation is by matching against a discrimination tree which is traversed in correspondance with TFS_l, from which types are ignored in a subtree reached by *path* where $\mathrm{omit}(path : v,\ TFS_l)$ applies and considered where $\mathrm{re-introduce}(path : v, TFS_l)$ applies. Where corresponding types from TFS_l and the tree are unequal, a new branch is grown, terminating with a new T_l: this mechanism ensures each terminal is marked. Because co-indexing by syntactic paths in a rule (Section 5) unifies *index*, this is re-introduced in an np and in a category that unifies with an np to form an np. The function in (3) is an example of a *restrictor*, Shieber (1985), although here we shall show that it does not lead to any approximation in parsing.

Precompilation generates multiple instantiations of each rule (1), paired with equivalent CFG rules, stored in a *tuple*:

$$\langle (T_0 \rightarrow T_1 \ldots T_n), rule-name \rangle \qquad (4)$$

Each represents the pair-wise unification of some $TFS_1 \ldots TFS_n$ with a sequence of TFSs where each is either lexical, like TFS_l, or derives from some rule instantiation, like TFS_0. For each sequence $T_1 \ldots T_n$ of categories derivable from such instantiations, using the discrimination tree, only the first corresponding instantiation encountered is treated, since all generate the same T_0.

By these means the closure becomes a bounded operation. This approach is valid since slashes are treated separately during parsing, and as in GPSG, Gazdar et. al. (1985), no phrase acceptable on syntactic grounds is then rejected on semantic grounds.

The CFG rules (4) are treated by a parser with a chart containing constituents :

$$\langle start, end, T_l, sem_l, slashes_l \rangle \text{ and}$$

$$\langle start, end, T_0, sem_0, slashes_0 \rangle \qquad (5)$$

The latter requires a CFG rule $T_0 \rightarrow T_1 \ldots T_n$ and consecutive constituents:

$$\langle start_1, end_1, T_1, sem_1, slashes_1 \rangle$$

$$\cdots \langle start_n, end_n, T_n, sem_n, slashes_n \rangle$$

where for $j > 1$, $start_j = end_{j-1}$

Slashes are treated by precompiling a table with entries $\langle T_s, T \rangle$ where T_s represents T_l or T_0, and T is the category assigned to a slash taken from a lexical TFS, and where $retrieve(T_s)$ unifies with $retrieve(T)$ prefixed by SYNSEM : LOC. Where in (4) $rule-name = filler-head$, (5) is formed only

$$\text{if } \exists \langle T_s, T \rangle \bullet (T_s = T_1 \wedge T = T_{s2} \in slashes_2) \qquad (6)$$

This mimics the operation of the $filler-head$ schema. In this case, in (5),

$$slashes_0 = slashes_1 \cup slashes_2 - \{T_{s2}\}$$

whereas with other schemas $\{T_{s2}\} = \varnothing$: this mimics the operation of the *non-local-feature principle*. The CFG grammar generated by this method accepts the same strings as the HPSG grammar and does not just approximate it.

The semantic component sem_0 represents a TFS, identical to sem_j in that sub-constituent known as the *semantic head*, according to the *semantics principle* of HPSG, except that some paths are co-indexed with paths in the semantic components of the other sub-constituents:

$$sem_0 = f(sem_1, \ldots, sem_n) \qquad (7)$$

To reduce copying overheads and eliminate TFS unification, sem_0 is encoded in a skeleton form, which is a Prolog structure:

$$s\_n(sem\_type_0, [p_1, \ldots, p_m]) \qquad (8)$$

The type sem_type_0 is not equal to T_0, and is the category of the lexical source of sem_0, which is a *transitive semantic head* of the corresponding TFS_0. For example, the sentence TFS contains a semantic component that derives from the head verb. In parsing, $p_k, 1 \leq k \leq m$ is bound to the unique identifier of the sub-constituent containing a path that must be co-indexed with a path in sem_j. In the prototype parser this path is the *k*th. one during a traversal of sem_j in the lexical source, that is found to be co-indexed with a syntactic path, that is in turn found to be prefixed by a path co-indexed between the corresponding daughters of the original rule (1) forming TFS_0 (see Section 5).

At retrieval time,

$$sem_0 = sem(sem\_type_0, [p_1, \ldots, p_m]) \quad (9)$$

where this indicates the semantic sub-structure in $retrieve(sem\_type_0)$, with the addition that each path for co-indexing associated with $1 \leq k \leq m$ is co-indexed with the appropriate path in the TFS given

$$\text{by } sem(sem\_type_{pk}, [p_1, \ldots, p_l]), \qquad (10)$$

from the p_k th. sub-constituent. A *retrieval graph* is compiled at category allocation time to support *retrieve*. Currently the semantic details of the slash are not recovered: sentences with slashes are accepted identically to ALE.

3 Modification of ALE rules

Figure 1 is the *head-subject-complement* schema in the ALE formalism. Figure 2 shows sections of the single Prolog clause containing 57 goals in 61 lines including the head, produced when ALE compiles the schema. This is invoked by the ALE chart parser after choosing the first of a speculative edge sequence. Its TFS is the structure *SVs*, which resembles (2): the functor represents the type, and each argument is a *value* from a (*feature: value*) pair. The *feature* is retrieved by successive instantiations of an ALE clause compiled from the grammar definition:

approps(*+type*, (*-approp_feature*:
-approp_type)) (10)

```
(phrase,Mother,synsem:loc:(cat)
              :(spr:[],subj:[],comps:[]))   M
    ===>
cat> word,HeadDtr,synsem:loc:(cat):
              (head:inv:plus,
               spr:Spr,
               subj:[SubjSynsem],
               comps:CompSynsems)),          D1
goal> (list_sign_to_synsem(CompDtrs,
                        CompSynsems)),       P1
cat>  (SubjDtr),                             D2
goal> (sign_to_synsem(SubjDtr,
                        SubjSynsem)),        P2
cats> (CompDtrs,ne_list),                    D3
goal> head_feature_principle(Mother,
                           HeadDtr),         P3
    semantics_principle(Mother,HeadDtr),     P4
    marking_principle(Mother,HeadDtr),       P5
    nonlocal_feature_principle(Mother,
         HeadDtr,[SubjDtr|CompDtrs])).       P6
```

Figure 1: The Head_subject_complement Schema from HPSG

Goals treat a TFS in a structure *Tag-SVs*, to allow type-coercion during unification into a sub-type, possibly supporting additional (never fewer) features. *Tag*, originally unbound, is then bound to a new copy: argument values are appropriate types (10) for new features, and unchanged for existing ones. Goal 21 references the second edge in the sequence, corresponding to *D2* of the original schema.

```
rule(Tag, SVs, Iqs, Start, End,Edge_no) :-      1
  add_to_type_word(Tag-SVs, Iqs, Iqs_out0),     2
  ud(A-bot, Tag-SVs, Iqs_out0, Iqs_out1),       3
  featval_synsem(Tag-SVs, FS2, Iqs_out1,
                               Iqs_out2),       4
      .     .     .     .     .
  featval_inv(FS5, FS6, Iqs_out5, Iqs_out6),    8
  add_to_type_plus(FS6, Iqs_out6, Iqs_out7),    9
  featval_subj(FS4, FS7,Iqs_out7, Iqs_out8),   10
  featval_hd(FS7, Tag2-SVs2, Iqs_out8,
                              Iqs_out9),       11
  ud(B-bot, Tag2-SVs2, Iqs_out9,
                              Iqs_out10),      12
  featval_tl(FS7,FS8,Iqs_out10, Iqs_out11),    13
  add_to_type_e_list(FS8, Iqs_out11,
                             Iqs_out12),       14
  featval_comps(FS4, Tag3-SVs3, Iqs_out12,
                             Iqs_out13),       15
  ud(C-bot, Tag3-SVs3, Iqs_out13,
                             Iqs_out14),       16
      .     .     .     .     .
  ud(D-bot, E-bot, Iqs_out15, Iqs_out16),      18
  ud(C-bot, F-bot, Iqs_out16, Iqs_out17),      19
  solve(list_sign_to_synsem(E-bot,F-bot),[],
              Iqs_out17, Iqs_out18),           20
  edge(Edge_noB,End,EndB,TagB,SVsB,
              IqsB,DaughtsB,RuleB),            21
      .     .     .     .     .
  deref(I, bot, Tag4, SVsList),                31
  SVsList=..[Type| MemList],                   32
  match_list_rest(Type,MemList,EndB,
         EndEdges,Edge_nos,[],Iqs_out27,
                             Iqs_out28),       33
      .     .     .     .     .
  solve(head_feature_principle(K-bot,L-bot),   46
     [semantics_principle(M-bot,N-bot),        47
      marking_principle(O-bot,P-bot),          48
      nonlocal_feature_principle(Q-bot,R-bot,
        S-bot)], Iqs_out40,Iqs_out41),         49
  add_to_type_phrase(Z-bot, Iqs_out41,
                             Iqs_out42),       50
      .     .     .     .     .
  add_edge_deref(Start,EndEdges,Z,bot,
    Iqs_out52,[Edge_no,Edge_noB|Edge_nos],
    head_subject_complement).                  61
```

Figure 2: The Head-subject-complement Schema after Compilation by ALE

Goal 33 corresponding to *D3* references the remaining edges: their number equals the number in the *comps* list of the sub-constituent unifying with the head daughter *D1*. Goal 61 creates the edge of the new phrase and corresponds to the mother *M*. Goal 20, and lines 46 to 49 forming one goal, are invocations of ALE procedures, corresponding to *P1*, and *P3* to *P6*. Apart from the first, these lines invoke HPSG principles.

ALE supports inequalities which HPSG does not use, here in lists with names of the form *Iqs_**: the output from one goal is input to the next, and edges contribute new lists for concatenation. The following three goals enforce constraints in the schema:

ud(*Tag1-SVs1, Tag2-SVs2, Iqs_in, Iqs_out*),
featval_FEAT(*FS_in, FS, Iqs_in, Iqs_out*),
add_to_type_TYPE(*FS_in, Iqs_in, Iqs_out*)

The first invokes a general purpose procedure for full-scale unification of the TFSs in its first two arguments. Often this just generates a co-indexing variable for reference elsewhere: *A* in goal 3 corresponds to *HeadDtr* in *D1*. *A* is an unbound *Tag* and *bot* is the most general type in the hierarchy: ud makes *A* reference *Tag-SVs*. If *SVs* becomes subject to type coercion, *A* references the new structure through *Tag*.

The second returns in *FS* the value of FEAT from *FS_in*, type-coercing this when FEAT is not appropriate. The add_to_TYPE goal obtains the common sub-type of TYPE and the type of *FS_in*, which is coerced to adopt this sub-type: the procedure is precompiled from the type hierarchy. Goals 4 to 9 use featval_FEAT and add_to_type_TYPE to enforce a *(path: value)* constraint in the first line of *D1*. Goals 10 to 16 treat three other constraints in *D1*: two of the values are co-indexing variables. For conciseness, the figure omits such goals after the first edge reference. Goals 31 and 32 extract the list of synsem structures *CompDtrs*, returned by the ALE procedure list_sign_to_synsem: the list derives from the value of the list *CompSynsems*.

Goal 50 coerces the initial type of *Z-bot*, the new constituent, to *phrase* as required in *M*. Then unshown goals constrain paths in this TFS according to the *(path: value)* constraints in *M*. Goal 61 invokes a procedure that asserts a new edge containing this coerced TFS referenced by *Z*, between positions *Start*, and *EndEdges* from the last edge of goal 33. The new edge contains a list of edge identifiers in the sequence, and the schema name, from the last two arguments.

The schema of Figure 2 is extended by our precompiler, to generate the tuples in a closure and details of co-indexing in order to guide semantics treatment. The modified goals are shown in Figure 3: clause modification is easier than modifying the complex compiler code in ALE, and the compiled schema already invokes ALE procedures appropriately.

During the closure, procedures invoked by goals 21 and 33 must constrain rule application so each sequence of edges is treated just once by each rule. Prolog backtracking cannot be altered to achieve this, and the edge and match_list_rest goals are modified. A list of identifiers of edges already invoked is passed between instances of these goals.

Detection of co-indexing requires access to the TFS in each sub-constituent after constraint application, and to the new TFS inside add_edge_deref before edge assertion, when co-indexing information is lost in copying. Since each schema is applied without backtracking to a single sequence of edges each containing *Tag-Bot*, the list of sub-constituent TFSs can be returned through the head of the rule: the new TFS, *Z-bot* appears as two arguments of the last goal.

```
rulejcb(Tag,SVs,Iqs,Start,End,Edge_no,
   - [Tag-SVs,TagB-SVsB,MemList],
   - Z-bot, + Edge_countA,
    - head_subject_complement,  + Edges_in,
     -Edges_outC):-                           1
edgejcb(Edge_noB,StartB,EndB,TagB,SVsB,
     IqsB,DaughtsB,RuleB,Edge_countA,
     Edge_countB, Edges_in, Edges_outB),  21
match_list_restjcb(Type,MemList,EndB,
     EndEdges,Edge_nos,[],Iqs_out27,
     Iqs_out28,Edge_countB, Edge_countC,
     Edges_outB, Edges_outC),             33
replace_add_edge_deref(Start, EndEdges,
                         Z, bot, Iqs_out52,
           [Edge_no, Edge_noB | Edge_nos],
            head_subject_complement )      61
```

Figure 3: The Head_subject_complement Schema after Further Compilation

The extra argument 7 of rulejcb is a list of sub-constituent TFSs after constraint application. *Z-bot* is the new TFS. *Edge_countA* is an initial count of 1 of the edges encountered so far. Remaining arguments are the rule name, and lists of edge identifiers before and after rule application.

Goal 21 invokes a new clause that invokes edge, and adds 1 to *Edge_countA* to form *Edge_countB*. The edge number, *Edge_noB*, is added to the head of *Edges_in*, to form *Edges_out*. Goal 33 invokes a recursive clause which similarly treats elements of *MemList*.

Goal 61 invokes a new procedure without additional arguments, to assign T_0 to the new TFS, *Z-bot,* using the discrimination tree and to assert a tuple (4). If T_0 is new, a fully dereferenced *Z-bot* is added to the retrieval graph, and a new edge asserted with the retrieved TFS. Another asserted clause associates T_0 with the edge number: similar clauses were asserted in lexical edge creation. They are referenced in tuple formation, from edge numbers in argument 6.

For debugging, an asserted clause contains the string deriving a tuple, structured into sub-phrases using brackets. Only the first sub-phrase deriving each category is used, to restrict numbers. Even so, this permitted the detection of over-generation arising from ungrammatical strings. This arose from the verbs *is, can, be, seem,* and the infinitival *to* not specifying their subject beyond co-indexing it with the subject of their complement, which is variously another (sometimes infinitival) verb or a predicate. This allowed generation of infinite sequences like “*X is is…*” where *X* is any phrase. It was cured by hand-specifying each subject as np. Complements were similarly treated for *believe* and *expect*. An automatic approach would propagate possible subjects, including alternatives to np in a larger grammar, from the complement into the outer verb.

To allow for large numbers of phrasal edges, edges optionally have unbound *SVs* arguments which are bound using the retrieval graph when referenced With over-generation cured, only 18,000 tuples were generated, and the facility was unused.

4 The Closure Algorithm

Using the TFS of each lexeme, an edge is asserted with an identifier between *Bottom* and *Last_free_edge_number-1*: *Start* and *End* are unbound so every sequence of edges is considered by rulejcb. Then the algorithm of Figure 4 is executed. On each recursive invocation, numbered in argument three, repeat_apply makes a pass over these edges and new edges generated in previous invocations. The first unused identifier after a pass is asserted in last_free_edge_number, and when this is unchanged after a pass in which no edges were asserted, termination occurs.

The first edge in a sequence is selected by apply_schemas_to_a_first_edge: *N* is the identifier which is incremented on each recursion between *Bottom* and *End*. If *SVs* in an edge is unbound, it is re-formed from the retrieval graph by add_SVs_to_edge.

Selection of the remaining edges in a sequence occurs non-deterministically within the compiled grammar rules. These are invoked by try_all_rules which references rulenames, previously asserted to identify all rules in a list. It invokes try_all_rules/4 to recurse through the list, each time invoking try_all_edges inside a negation, since a failure-driven loop treats multiple edge sequences by invoking the non-deterministic rulejcb. The TFS of the first edge appears in the first two arguments: its identifier *N* appears alone and as the first in a list of edge identifiers for the sequence, completed inside rulejcb.

To minimise compilation time, each sequence must be treated once by each rule. Sequences unpredictably contain 2 or 3 edges, and a first edge *E1* can combine with edges created by sequences of edges treated after *E1*. No edge can ever be discarded as a candidate for any daughter in any rule. Sequences are too numerous to record by asserting clauses to be referenced before rule application.

The chosen solution is straightforward and fairly efficient. On each pass of repeat_apply, a range of acceptable edge identifiers is established. Four global variables holding identifiers, *Bottom*, *Top*, *End* and *Last_free_edge_number* are asserted in clauses bottom, top, end and last_free_edge_number respectively. The first three are adjusted in repeat_apply, whilst the last is incremented in replace_add_edge_deref on edge assertion.

On any pass, edges with *N>End* are asserted, and apply_schemas_to_a_first_edge treats first edges between *Bottom* and *End*. At the end of a pass, *Top* is set equal to *End* and *End* is then set to *Last_free_edge_number-1*: before the first pass *Top is set to Bottom – 1*.

```
repeat_apply(Bottom,Last_free_edge_number,
                 Pass_no):-
  end(End),
  apply_schemas_to_a_first_edge(Bottom,End),
  last_free_edge_number(Last_free_edge_no2),
  End2 is Last_free_edge_number2 – 1,
  retractall(end(_)),assert(end(End2)),
  Pass_no_out is Pass_no + 1,
  (! , ( (not(Last_free_edge_number = =
                   Last_free_edge_number2))
               ->
     (repeat_apply(Bottom,Last_free_edge_no2,
            Pass_no_out),!)
                ;true
   ).
apply_schemas_to_a_first_edge(N,L):-
  edge(N,Start,End,Tag, SVs, Iqs, Dtrs,
        Rule_name),!,
  (var(SVs) -> add_SVs_to_edge(N,Tag,SVs)
                ; true),
  try_all_rules(Tag,SVs,N),!,
  New_N is N + 1,
  (not(New_N > L) ->
  (apply_schemas_to_a_first_edge(New_N,L))
                         ; true).
try_all_rules(Tag,SVs,N):-
  rulenames(Rulenames),
  try_all_rules(Tag, SVs,N,Rulenames).
try_all_rules(Tag,SVs,N,
                   [Rulename|Rulenames]):-
  not(try_all_edges(Tag, SVs,N,Rulename)),
  try_all_rules(Tag, SVs,N,Rulenames),!.
try_all_edges(Tag, SVs,N,Rulename):-
  rulejcb(Tag, SVs, [], _ , _ , N, Daughters,
              Mother,1,Rulename,[N],D2),
  fail.
```

Figure 4: Algorithm for Tuple Generation

Consequently, edges between *Top+1* and *End* were always created during the last pass: initially this is the set of lexical edges.

In edgejcb and match_list_restjcb, an edge identified by *d2* or *d3* depending on position in a sequence, has its *SVs* unified with the rule daughter if a following test succeeds. *Test 1* succeeds if *d1* was created on the previous pass (or as a lexical edge, for pass 1), and *d2* (and *d3* for a 3-daughter sequence), were created on any pass (including lexemes) up to and including the last. The *d1* restriction prevents treatment on multiple passes. If *Test 1* fails, *Test 2* succeeds if *d2* is newly created on the previous pass, and *d1*, (and *d3*), were created on any pass (including lexemes) up to and including the last. If this fails, *Test 3* is passed if *d1* and *d2* were created on any pass up to but not including the last pass, and if the same rule successfully treated them earlier as the start of a 3-edge sequence.

By delaying combination of a new edge into a sequence, until the pass after its creation, we avoid a repeat pass to catch the case where the other edges do not yet exist at creation time. Overall this necessitates < 1 extra, low cost pass. HPSG eliminates sequences mainly by constraints between daughters, avoided on this extra pass by trivial arithmetic comparisons in $O\left(\left(End-Top\right)\times\left(End-Bottom\right)\right)$ whilst the unavoidable $O\left(End-Top\right)$ costs of first-daughter unification are small

Test 3 treats a new *d3*, with old *d1* and *d2*, one of which must have been new on an earlier pass, when it was treated under *Test 1* or *2*. Since *d3* is a potential complement of either *d1* or *d2*, some sequence *[d1, d2, d3']* was treated earlier, even if *d3'* failed unification: at that time treated_edges_before(*Rulename, [d1,d2])* was asserted. Unification of *d2* in *[d1, d2, d3]* takes place (on repeated passes) only if this asserted edge is found. Successful unifications will be repeated, but the closure on our test grammar has only 7 passes, and in backtracking only one *[d1, d2]* unification takes place for all *[d1, d2, d3]*.

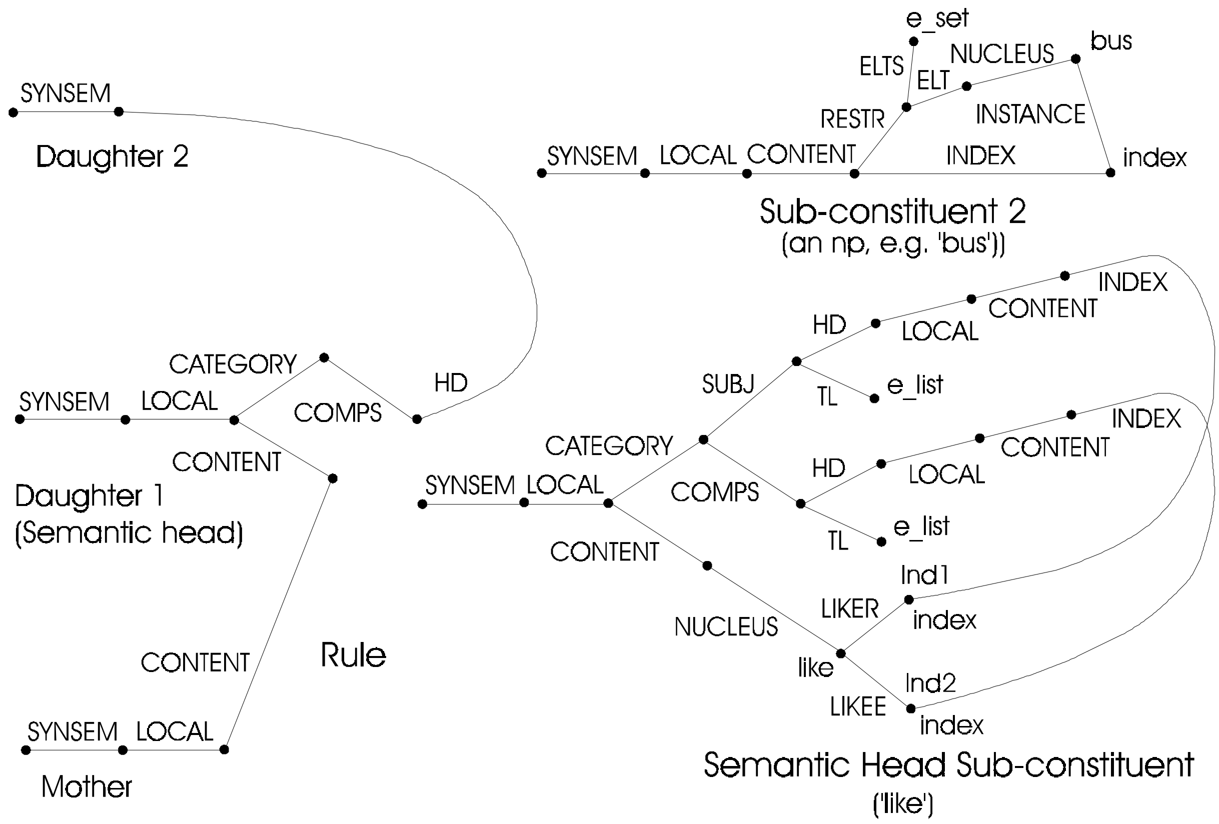


Figure 5: Application of Head-Complement Schema to an Edge Sequence

5 Treatment of Semantic Structures

In a constituent, the co-indexing of paths in a copy of the semantic sub-structure of a semantic head was explained in (7) to (10). Figure 5 illustrates this for the *head-complement* schema.. In the verb (the semantic head), the semantic and syntactic paths *Pm* and *Ps* co-index at the variable *Ind2*, where :

Pm = SYNSEM: LOCAL: CONTENT: NUCLEUS: LIKEE and

Ps = SYNSEM: LOCAL: CATEGORY: COMPS: HD: LOCAL: CONTENT: INDEX

The rule co-indexes a pair of syntactic paths in head and non-head daughters:

Ph = SYNSEM: LOCAL: CATEGORY: COMPS: HD and

Pnh = SYNSEM

Since the concatenation *(Ph : Psuffix) = Ps* where *Psuffix* = LOCAL: CONTENT: INDEX, then the TFS reached by *Pm* in the head unifies with that reached by *(Pnh: Psuffix)* in the other sub-constituent. The *index* structure reached by INDEX has a dual role in an np: as well as appearing in the semantic structure of the phrase, it also tests syntactic agreement of its *number*, *person* and *gender* features via co-indexing in the rule. It was therefore considered in category allocation.

Analysis involves applying each grammar rule to a sequence of *Tag-bot* structures. In these, the *Tag* of each co-indexed node is bound to a 3-digit integer *xyy*, where *yy* and *x* are ordinals identifying the co-indexed node in the rule, and the first daughter with a path to that node. The head daughter is distinguished since its path SYNSEM: LOCAL: CONTENT co-indexes with that path in the mother. This leads to <*Ph, Pnh*> pairs, each identified by an integer *Path_no*, for each rule. For each pair in each rule a clause is asserted: rule_paths(*Rulename, Path_no, _, Daught_no*) where *Daught_no* locates *Pnh*.

Each lexeme that a tuple (4) shows to be unifiable with the head daughter of some rule, is treated to detect 3-tuples of the form:

< *Indn, Path_no, Suffix_no* > according to the mechanism illustrated in the example above: *Suffix_no* identifies a single *Psuffix* path. A skeleton semantic structure (8) is derived,

where each p_k argument corresponds to a 3-tuple, ordered as *Indn* nodes are encountered in a TFS traversal. *Path_no* fields appear in the same order in an asserted clause r_n(*cat, Paths*) where *cat* is T_l allocated to the lexeme. During constituent construction in CFG parsing, *Daught_no* of the sub-constituent is known, and *Rulename* is deduced from the tuple (4), so r_n and rule_paths identify the p_k argument in (8) to bind to the edge identifier of the sub-constituent.

Retrieval graph arcs leading to nodes like *Ind2* are marked by asserted clauses:

arc_to_retrieval_tag2(+*Current_arc_no,*
+*Category, _ , -Path_no, -Suffix_no*).

For each possible <*Path_no, Suffix_no*> pair generating *(Pnh : Psuffix)*, this path is speculatively followed in the retrieval graph for each lexeme to identify a node and assert:

find_type_node_compiled(+*Path_no,*
+*Category,* +*Suffix_no,*
-*Type_node_no,* -*Arc_no_in*).

When retrieving the TFS_0 of a CFG constituent from the retrieval graph, once the semantic path *semantics* from (3) is encountered, the sem_type_0 of (8) is treated, starting at the node reachable by *semantics*. Where an arc number matches that in an arc_to_retrieval_tag2 clause, *Path_no* and *Suffix_no* are used to address find_type_node_compiled. *Category* here derives from sem_type_{pk} from (10), being the category of the semantic skeleton in a sub-constituent identified by the appropriate p_k in (8): the ordering ensures that successive arcs matching arc_to_retrieval_tag2 clauses correspond correctly with consecutive p_k arguments. The arc in the TFS being constructed is redirected to a copy of the indicated node, and traversal of the retrieval graph continues from that node. *Arc_no_in* is used recursively with arc_to_retrieval_tag2 to detect if the target type-node should itself be replaced by a node derived from a further sub-constituent.

This technique also properly treats the *head-subject-complement*, *subject-head* and *adjunct-head* rules. In this last case the RESTR set of identifiers for the noun and adjectives is properly constructed. This set is not accessible from the semantic TFS of the sentence since the grammar (probably incorrectly) co-indexes the *Indn* of the verb with the node reached by INDEX in the np, rather than with that from which the INDEX and RESTR arcs emerge. The *specifier-head* rule uses a different form of co-indexing which we have not yet treated: appropriate semantic structures are still returned for the sample grammar, where no specifier has a more specific *index* structure than any head np. The counter-example*"a sheep*" which derives its *number* from the specifier is not in the lexicon. Similarly, the approximation that an arc in the head is redirected to a node in another sub-constituent avoids unification of *index* structures to properly treat "*… sheep eat(s)":* such low-cost unification can easily be added to retrieval.

Our precompiled CFG is an exact equivalent of the TFS grammar rather than an approximation, since our restrictor does not eliminate paths affecting agreement except for *slash*: slashes are treated separately in our parsing algorithm, as in CFG. Since the schemas treated enforce agreement through a syntactic path in the head, the semantics in the head can be omitted by the restrictor. This also eliminates the major source of TFS expansion in a closure. The test grammar does not make the daughter TFSs of a phrase accessible except through its semantics, so these are also eliminated from consideration.

The path to the *index* structure in an np is co-indexed by the mechanism in Figure 5, and by other schemas, some of which co-index it with *index* in another sign combining with np to produce np. Therefore it is re-introduced for category allocation after semantics is excluded (3). The RESTR component is still excluded: a linguistic reason is that its value depends on word morphology, whilst syntactic agreement depends on more general features.

However, no automatic mechanism could restrict the *index* treatment as we do. Verbs like *believe*, *seem*, *persuade*, *expect* and *promise* take a vp or an s as complement. They could potentially specify agreement with the semantic part of a vp of varying saturation. In practice only the syntactic HEAD of the complement is constrained, so these verbs differ from verbs that take np as a subject. A linguistic reason is that the semantic structure of a verb depends on its word morphology as in Figure 5, whilst in an np this dependency applies only to RESTR and not to INDEX.

An automatic mechanism to generate our restrictor would necessitate a closure from a sample of the lexicon, since only when syntactic agreement occurs can the need for semantic agreement be assessed. The linguist can predict such agreement by inspection, so a better approach might be to automatically generate diagrams like Figure 5 to guide in the choice of restrictor. An over-drastic restrictor becomes apparent only when a retrieved TFS from CF parsing does not match the original from TFS parsing. Automatic mechanisms for comparison might be worth investigating.

Automatic mechanisms to derive our treatment of slashes may be possible, since they are associated with lists that grow during parsing, not shrink as do *subcat*, *subj* or *comps* lists. Our test grammar does not maintain quantifier lists, but their behaviour is in many ways similar to that of slashes.

We do not currently retrieve the semantic structure of slashes. If each slash is paired with the edge number of its lexical origin, and details of slashes satisfying the *filler-head* schema during parsing are indexed by edge number, then the semantics can be retrieved from the TFS corresponding to T_1 in (6), when the slash is encountered during retrieval.

6 Conclusion

It has proved practical to precompile in an acceptable time a realistic HPSG grammar into exactly equivalent (neglecting semantics) CFG categories and rules, of reasonable number and compact size, together with a table to control slash agreement. It was also possible to generate data structures for building skeleton semantic structure and retrieving its full structure after parsing, obviating the need for TFS unification. A Prolog prototype parses 18 times faster, and is estimated to be 180 times faster in an optimum imperative code solution. This predicted speed-up would exceed that obtained with a CFG approximation, where TFS unification must follow CFG parsing. The kind of co-indexing used in the *specifier-head* schema is not treated in semantic retrieval, but the method seems extensible to embrace this.

Acknowledgements

This research was funded by a legacy from Miss Nora Brown, and Workshop attendance funded by INVU. Our thanks to the anonymous referees and to Mr. Stephan Oepen for their suggestions, and to Bernd Kiefer and Kentaro Torisawa for copies of their papers.

References

Brown, J.C. and Manandhar, S. (2000) *Compilation versus Abstract Machines for Fast Parsing of Typed Feature Structure Grammars*, Future Generation Computer Systems 16, pp. 771-791.

Carpenter, B. and Penn, G. (1996) *Compiling Typed Attribute-value Logic Grammars*, in “Recent Advances in Parsing Technology”, H.Bunt, M. Tomita, ed., Kluwer Academic Press, Dordrecht, pp. 145-168.

Carroll, J.A. (1993) *Practical Unification-based Parsing of Natural Language*, Technical Report No. 314, University of Cambridge Computer Laboratory.

Gazdar, G., Klein, E., Pullum, G., Sag, I. (1985) *Generalized Phrase Structure Grammar*, Blackwell, Oxford.

Kiefer, B. and Krieger, H.-U. (2000) *A Context-Free Approximation of Head-driven Phrase Structure Grammar*, in Proceedings of the 6th. Int. Workshop on Parsing Technologies (IWPT), Trento, Italy, pp. 135-146.

Matheson, C. (1996) *Developing HPSG Grammars in ALE*, Course Notes, Human Communications Research Centre, University of Edinburgh. http://www.ltg.hcrc.ed.ac.uk/projects/ledtools/ale-hpsg/index.html.

Pollard, C. and Sag, I.A. (1994) *Head-Driven Phrase Structure Grammar*, University of Chicago Press, Chicago.

Shieber, S.C. (1985) *Using Restriction to Extend Parsing Algorithms for Complex Feature Based Formalisms*, in Proceedings of the 23rd. Annual Meeting of the Association for Computational Linguistics, pp. 145-152.

Torisawa, K., Nishida, K., Miyao,Y., and Tsujii, J.-I. (2000) *An HPSG Parser with CFG Filtering*, Natural Language Engineering 6 (2), pp. 1-18.

Time as a Measure of Parsing Efficiency

Robert C. Moore
Microsoft Research
One Microsoft Way
Redmond, Washington 98052, USA
bobmoore@microsoft.com

Abstract

Charniak and his colleagues have proposed implementation-independent metrics as a way of comparing the efficiency of parsing algorithms implemented on different platforms, in different languages, and with different degrees of "incidental optimization". We argue that there are easily immaginable circumstances in which their proposed metrics would mask significant differences in efficiency; we point out that their data do not, in fact, support the usability of such metrics for comparing the efficiency of different algorithms; and we analyze data for a similar metric to try to quantify the degree of variation one might expect between such metrics and actual parse time. Finally, we propose a methodology for making cross-platform comparisons through the use of reference parser implementations.

1 Introduction

The title "Time as a Measure of Parsing Efficiency" may seem highly tautologous, since in computer science "efficiency", unless otherwise qualified, is usually taken to mean speed of execution. However, Charniak and his colleagues (Caraballo and Charniak, 1998; Charniak, Goldwater, and Johnson, 1998; Blaheta and Charniak, 1999) have argued for another metric—edges popped off the agenda of a chart parser—as being platform independent. Now Roark and Charniak (2000) propose a related measure, "events considered", that is applicable to a wider range of approaches.

The present paper attempts to make the case for going back to time as the primary measure of parsing efficiency. First, we explore some general issues concerning the type of metrics proposed by Charniak and his colleagues. Then we demonstrate using empirical data that implementation-independent measures similar to that proposed by Charniak can vary in how well they correlate to execution time by as much or more than the improvements reported by Charniak et al. in some of their experiments. Finally, we propose a methodology for comparing parse times on different platforms, through the use of reference parser implementations.

2 Implementation-Independent Parser Metrics

As mentioned above, Charniak and his colleagues (Caraballo and Charniak, 1998; Charniak, Goldwater, and Johnson, 1998; Blaheta and Charniak, 1999; Roark and Charniak, 2000) have argued for implementation-independent measures of parsing efficiency, including edges popped off the agenda of a chart parser and, currently, "events considered", which is defined by Roark and Charniak (2000) as "the number of 'events', however they are defined for a particular parser, for which a probability must be calculated, in order to find the parse." It is argued in these papers that such metrics enable parsing efficiency to be compared at the algorithmic level without being concerned with the degree of low-level optimization that has been performed.

Roark and Charniak have applied the events-considered metric both to a best-first parser and to a word-synchronous beam-search-based parser. These parsers differ in many respects, but have a number of attributes in common that bear on the current discussion:

- They both are probabilistic, and seek to find the parse with the highest probability according to some model.
- They are both heuristically pruned. That is, they do not explore all analyses that

have non-zero probability, nor are they even guaranteed to explore all analyses that might turn out to have the highest probability.

- They both prune partial analyses on the basis of a figure of merit that can be viewed as based on an heuristic estimate of the expected probability that the full model would assign to extensions of a given partial analysis.

Within this framework, let us consider some of the ways that the number of events considered could fail to correlate with parse time in an essential way; that is, not based on what Roark and Charniak refer to as "incidental optimizations". First, the full probability models might take vastly different amounts of time to compute. For example, maximum-entropy, or exponential, models (Berger, Della Pietra, and Dell Pietra, 1996) are believed by some to hold the promise of significantly higher accuracy in a variety of tasks than the more conventional, simpler models that Charniak and colleagues have used. Unfortunately, maximum entropy models are much more expensive to compute than these simpler models, if they are normalized to produce true probability distributions.[1] Thus if we compared a parser that used a normalized exponential model to compute the probability of exactly the same events that one of Roark's and Charniak's parsers considers, we would expect to find it much slower than theirs. This might be a trade-off worth making, if it delivered better parsing accuracy, but it would be ludicrous to claim it was equally efficient, simply because it considered no more events than Roark's and Charniak's parsers.

Another way that the events-considered metric might fail to correlate with parse time concerns the figure of merit used to prune he search. The efficacy of pruning is perhaps the most important determinant of parsing efficiency in the types of parsers considered by Roark and Charniak. The better the figure of merit is as a predictor of the probability assigned by the full model to the best extension of a partial analysis, the smaller the number of partial analyses that need to be extended to find the best full analysis. It is easy to imagine, however, that some "better" figure of merit might take more time to compute than it repays in reducing the search space. Suppose a there is a sophisticated figure of merit that allows reducing the number of events considered by half over a cruder figure of merit, but it takes so long to compute that it increases the amount of time per event considered by a factor of ten. The events-considered metric will rate the sophisticated figure of merit as twice as good, even though in reality a parser that used it would be five times slower. This is a classic problem in heuristic search and has been discussed extensively in the literature on computer chess and automatic theorem proving.

3 Empirical Evaluation

The arguments in the preceding section are in a sense speculative, since they discuss possibilities that might occur, but we have not demonstrated that they do occur in practice. For that, empirical evidence is required. At first blush, Roark and Charniak might seem to have produced empirical evidence to the contrary, since they present graphs of impressively linear relationships between events considered and time. These linear relationships, however, are demonstrated only with respect to one parser at a time, where the only thing that is varied is the degree of pruning. Thus, although Roark and Charniak argue for events-considered metric as a way of comparing very different parsing algorithms, they demonstrate only that it correlates well with efficiency for essentially identical algorithms, where only a single parameter is varied.

We have recently carried out a series of experiments (Moore, 2000) comparing the efficiency of several variants of left-corner parsing with a number of other well-known context-free parsing algorithms. In contrast to the studies of Charniak et al., these experiments looked at non-stochastic algorithms, and the comparisons are made on the basis of exhaustive, rather than heuristically-pruned, parsing. Thus we cannot directly apply either Roark and Charniak's events-considered metric, or the earlier edges-popped-of-the-agenda metric, to our results. However, the total number of edges in the chart, which we do have data on, is a very similar sort of implementation-independent metric, and is often used informally to judge parser ef-

[1]Note that Charniak (2000) has also explored *unnormalized* exponential models, which are fast to compute.

ficiency. (Arguably, this would be the same as the edges-popped-of-the-agenda metric, when parser is allowed to run to exhaustion.)

To evaluate the suitability of using total number of edges in the chart as an efficiency metric, we will present a selection of results from our CFG parsing experiments, including edge statistics not previously published. Results for five parsing algorithms on three different grammars are included. The parsing algorithms consist of two variants of left-corner parsing (LC_1+BUPM and LC_2+BUPM), an Earley/Graham-Harrison-Ruzzo parser (E/GHR), a Cocke-Kasami-Younger parser (CKY), and a generalized LR parser without look-ahead (GRL(0)). (The identifiers for these parsers are the ones used in our earlier report.) It should be mentioned that all these parsers represent the best of several implementations of the general approach, and all parsers are implemented using similar techniques and data structures wherever possible. Furthermore, all algorithms are implemented in the same language (Perl5) on the same platform (Windows 2000, 550 MHz Pentium III). Thus we believe that the performance differences are genuinely representative of inherent differences in the algorithms, and not just irrelevant implementation details.

The grammars used are independently motivated by analyses of natural-language corpora or actual applications of natural language processing. The CT grammar was compiled into a CFG from a task-specific unification grammar written for CommandTalk (Moore et al., 1997), a spoken-language interface to a military simulation system. The ATIS grammar was extracted from an internally generated treebank of the DARPA ATIS3 training sentences. The PT grammar is Charniak's PCFG grammar extracted from the Penn Treebank, with the probabilities omitted. The most significant variation among the grammars is the degree of ambiguity of the test sets associated with each grammar. The CT test set has 5.4 parses/sentence with the CT grammar, the ATIS test set has 940 parses/sentence with the ATIS grammar, and the PT test set has 7.2×10^{27} parses/sentence with the PT grammar.

Table 1 shows the results of applying these five parsers to the three grammars and their associated test sets. The first column gives the average number of chart edges per sentence, including both complete and incomplete edges (where incomplete edges are generated). For the GLR(0) parser, this is equivalent to the number of edges in the graph-structured stack used by most implementations of GLR parsing. The second column gives the average number of seconds per sentence to parse exhaustively. This includes only time to populate the chart, and does not include time to extract parses. The final column compares the second column to the first, to derive an average number of milliseconds per chart edge. The more constant this number is across the different parsers and grammars, the better total edges in the chart will be as a measure of parser efficiency.

If we look at the last column in detail, we see that total number of chart edges generated does have some crude validity as a measure of parsing efficiency; since the majority of the test cases fall around 0.1 milliseconds per edge. However, the variation is fairly large. The two left-corner parsers make a particularly interesting comparison, because they differ only in a single detail, and produce exactly the same edges. In these parsers incomplete edges are subjected to two tests before being added to the chart. The mother of an incomplete edge has to be a possible left corner of the next daughter required by some previous incomplete edge at the appropriate position in the input; furthermore, the next daughter of the incomplete edge being tested has to have the next token in the input as a possible left corner. These tests are independent, so they can be performed in either order. In LC_1+BUPM the check on the mother is performed first, and in LC_2+BUPM the check on the next daughter is performed first. These results show that performing the check on the mother first is 14% to 68% slower than performing the check on the next daughter first. This is a substantial difference that cannot be detected looking only at the edges added to the chart.

There are several places in the data where the numbers of chart edges strongly, but incorrectly, predict which of two parsers should be faster on a given grammar. For example, the LC_1+BUPM parser generates only about half as many edges as the E/GHR parser with the ATIS grammar, but is nevertheless 35% slower.

Grammar	Parser	Edges/sent	Sec/sent	msec/edge
CT	LC_1+BUPM	165.3	0.0219	0.132
	LC_2+BUPM	165.3	0.0191	0.116
	E/GHR	283.0	0.0448	0.158
	CKY	1598.2	0.1540	0.096
	GLR(0)	159.0	0.0214	0.135
ATIS	LC_1+BUPM	673.4	0.119	0.177
	LC_2+BUPM	673.4	0.071	0.105
	E/GHR	1276.6	0.088	0.069
	CKY	537.3	0.078	0.145
	GLR(0)	1282.5	0.143	0.112
PT	LC_1+BUPM	6675.4	1.14	0.171
	LC_2+BUPM	6675.4	0.90	0.135
	E/GHR	11143.9	0.92	0.083
	CKY	5785.6	1.70	0.294
	GLR(0)	51285.9	28.86	0.563

Table 1: Chart edge and time statistics for CFG parsing algorithms

It is also notable that the CKY and GLR(0) parsers are not even self-consistent in terms of time per edge across grammars. They take substantially more time per edge on the PT grammar than on the other two.

One way to quantify the uncertainty about the relationship between edges produced and parse time is to use statistical analysis to estimate how many fewer edges one parser has to produce to be reasonably certain that it is actually faster than another. We have performed a crude version of such an analysis on the data in Table 1, under the assumption that they represent a random sample of parsers and grammars, suggesting the following: To be 99% certain that one parser is faster than another, it must produce about 80% fewer edges; to be 95% certain, it must produce about 70% fewer edges; and to be 90% certain, it must produce about 60% fewer edges. This analysis undoubtedly suffers from the paucity of the data, and performing a wider range of experiments might well give us tighter bounds, but it at least represents a first cut at quantifying the variation in time-per-edge across different parsing algorithms and grammars. (See the Appendix for information on how the analysis was performed.)

If we look at Charniak's latest paper reporting improved parser efficiency (Blaheta and Charniak, 1999), we find a reported reduction in edges popped off the agenda of about 60%. We do not doubt that this result is in fact a substantial improvement over the previous method it was compared to, but in the context of the variation in time per edge among the parsers presented here, we would be hard pressed to claim this result represents a statistically significant improvement in parsing speed, without seeing parsing times not provided by Blaheta and Charniak.

4 Towards a Cross-Platform Methodology

The substantial variation in time per edge among the parsers and grammars discussed here strongly suggests that, in making efficiency comparisons, actual parse times should be measured whenever that would be meaningful. For a particular research team reporting incremental progress of a continuing research program, this hardly seems too much to ask.

For cross-research-group comparisons the matter is more problematical. How is one to compare a parser written in Lisp running on an 400 MHz UltraSPARC processor to one written in C++ running on a 750 MHz Pentium III? The answer proposed here is to create a set of reference parser implementations, in a variety of languages, coupled with reference grammars and test sets. Obviously, for different classes of grammar—unification grammars, CFGs, and PCFGs—meaningful comparisons remain difficult; but for a given class, this approach should make cross platform comparisons straightforward. For example, for CFGs, a particular variant of CKY could be chosen, and implemented as efficiently as possible in C, Lisp, Prolog, Perl(!), and any other language considered relevant. The source code for implementations would need to be provided, so that the claim to be the best possible implementation of the algorithm in the language could be examined, and improvements made, without changing the basic algorithm. Then, whenever anyone claims to have an improved algorithm for CFG parsing, written in any of the supported languages and any platform that supports the language, they could meaningfully measure how much faster their algorithm is than CKY on the standard grammars and test sets, by timing their parser against the reference CKY parser for their platform. Similarly, reference parser implementations could be created for PCFGs and unification grammars to facilitate cross-research-group studies of parser efficiency for those formalisms.

5 Conclusions

Charniak and his colleagues have proposed implementation-independent metrics as a way of comparing the efficiency of parsing algorithms implemented on different platforms, in different languages, and with different degrees of "incidental optimization". We have tried to argue that there are easily immaginable circumstances in which their proposed metrics would mask significant differences in efficiency; we have pointed out that Roark and Charniak's data do not, in fact, support the usability of their metric for comparing the efficiency of different algorithms; and we have analyzed data for a similar metric (chart size) to try to quantify the degree of variation one might expect between such metrics and actual parse time.

Finally, we have proposed a methodology for making cross-platform comparisons through the use of reference parser implementations. This methodology does not, unfortunately, address Roark and Charniak's desire for a metric that is insensitive different degrees of incidental optimization. Indeed, in our own work comparing different parsing algorithms we have tried to be scrupulous in comparing implementations that are as similar as possible in that regard. In effect, Charniak et al. seem to be seek metrics which obviate the need for controled experiments, but in the end, there is no substitute for them.

References

Berger, A., S. Della Pietra, and V. Della Pietra (1996) "A Maximum Entropy Approach to Natural Language Processing," *Computational Linguistics*, Vol. 22, No. 1, pp. 39–71.

Blaheta, D., and E. Charniak (1999) "Automatic Compensation for Parser Figure-of-Merit Flaws," in *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, College Park, Maryland, pp. 513–518.

Caraballo, S., and E. Charniak (1998) "New Figures of Merit for Best-First Probabilistic Chart Parsing," *Computational Linguistics*, Vol. 24, No. 2, pp. 275–298.

Charniak, E., S. Goldwater, and M. Johnson (1998) "Edge-Based Best-First Chart Parsing," in *Proceedings of the Sixth Workshop on Very Large Corpora*, Montreal, Canada, pp. 127–123.

Charniak, E. (2000) "A Maximum-Entropy-Inspired Parser," in *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics*, Seattle, Washington, pp. 132–139.

Moore, R., et al. (1997) "CommandTalk: A Spoken-Language Interface for Battlefield Simulations," in *Proceedings of the Fifth Conference on Applied Natural Language Processing*, Association for Computational Linguistics, Washington, DC, pp. 1–7.

Moore, R. (2000) "Improved Left-Corner Chart Parsing for Large Context-Free Grammars," in *Proceedings of the Sixth International Workshop on Parsing Technologies*, ACL/SIGPARSE, Trento, Italy, pp. 171–182.

Roark, B., and E. Charniak (2000) "Measuring Efficiency in High-Accuracy, Broad-Coverage Statistical Parsing," in *Proceedings of the COLING 2000 Workshop on Efficiency in Large-Scale Parsing Systems*, Luxembourg.

Confidence Level	Method 1	Method 2
0.99	0.19	0.17
0.95	0.30	0.29
0.90	0.39	0.38

Table 2: Edge ratios for different significance levels

Appendix

The statistical analysis mentioned in Section 3 was performed by comparing the milliseconds/edge values for all pairs of parsing algorithms, for each grammar and test set. Specifically, we computed the t statistic for logarithms of ratios of milliseconds/edge values for sets of pairs of parsers applied to the same grammar and test set. We used logarithms of ratios rather than the ratios themselves, on the grounds that, *a priori* we would expect values of n and $1/..n$ to be about equally likely for this ratio.

We computed the t statistic in two different ways. Method 1 was simply to compute the t statistic for the set of logarithms of ratios of the milliseconds/edge values for all pairs of parsing algorithms, for each grammar and test set. This is not statistically valid, however, because the values we are computing the t statistic for are not independent, due to re-use of milliseconds/edge values in different pairwise comparisons. Essentially, we have manufactured 30 data points out of an original set of only 15 data points.

In Method 2, we maintained independence by randomly selecting pairs of parsers so that each parser was only counted once, applied to each grammar and test set. Since this only looks at a subset of the possible pairwise comparisons, we repeated this 1000 times, so that each possible pair would be selected approximately the same number of times, and averaged the results. We then used these t statistics to compute the ratios of edges required to be certain that the parser producing fewer edges would actually take less time than the other, at the 0.99, 0.95, and 0.90 confidence levels, using a single-tailed test (See Table 2). The two methods did not differ greatly in their results, except that Method 2 produced slightly greater uncertainty. This may have been simply due to the smaller number of data points used in each iteration in Method 2.

Measuring efficiency in high-accuracy, broad-coverage statistical parsing*

Brian Roark **Eugene Charniak**
Brown Laboratory for Linguistic Information Processing (BLLIP), and
Cognitive and Linguistic Sciences Computer Science
Box 1978 Box 1910
Brown University Brown University
Providence, RI 02912 Providence, RI 02912
brian-roark@brown.edu ec@cs.brown.edu

Abstract

Very little attention has been paid to the comparison of efficiency between high accuracy statistical parsers. This paper proposes one machine-independent metric that is general enough to allow comparisons across very different parsing architectures. This metric, which we call "events considered", measures the number of "events", however they are defined for a particular parser, for which a probability must be calculated, in order to find the parse. It is applicable to single-pass or multi-stage parsers. We discuss the advantages of the metric, and demonstrate its usefulness by using it to compare two parsers which differ in several fundamental ways.

1 Introduction

The past five years have seen enormous improvements in broad-coverage parsing accuracy, through the use of statistical techniques. The parsers that perform at the highest level of accuracy (Charniak (1997; 2000); Collins (1997; 2000); Ratnaparkhi, 1997) use probabilistic models with a very large number of parameters, which can be costly to use in evaluating structures. Parsers that have been built for this level of accuracy have generally been compared only with respect to accuracy, not efficiency. This is understandable: their great selling point is the high level of accuracy they are able to achieve. In addition, these parsers are difficult to compare with respect to efficiency: the models are quite diverse, with very different kinds of parameters and different estimation and smoothing techniques. Furthermore, the search and pruning strategies have been quite varied, from beam-search to best-first, and with different numbers of distinct stages of processing.

At a very general level, however, these approaches share some key characteristics, and it is at this general level that we would like to address the issue of efficiency. In each of these approaches, scores or weights are calculated for events, e.g. edges or other structures, or perhaps constituent/head or even head/head relations. The scores for these events are compared and "bad" events, i.e. events with relatively low scores, are either discarded (as in beam search) or sink to the bottom of the heap (as in best-first). In fact, this general characterization is basically what goes on at each of the stages in multi-stage parsers, although the events that are being weighted, and the models by which they are scored, may change[1]. In each parser's final stage, the parse which emerges with the best score is returned for evaluation.

We would like to propose an efficiency metric which we call *events considered*. An event is *considered* when a score is calculated for it. Search and pruning techniques can be judged to improve the efficiency of a parser if they reduce the number of events that must be considered en route to parses with the same level of accuracy. Because an event must have a score for a statistical parser to decide whether it should be retained or discarded, there is no way to improve this number without having improved either the efficiency of the search (through, say, dynamic programming) or the efficacy of the pruning. We will argue that this is not the case with

* Thanks to everyone in the Brown Laboratory for Linguistic Information Processing (BLLIP) for valuable discussion on these issues. This research was supported in part by NSF IGERT Grant #DGE-9870676, and NSF LIS Grant #SBR-9720368.

[1]Even within the same stage, events can be heterogeneous. See the discussion of the EC parser below.

competitor measures, such as time or total heap operations, which can be improved through optimization techniques that do not change the search space. This is not to say that these techniques do not have a great deal of value; simply that, for comparisons between approaches to statistical parsing, the implementations of which may or may not have carried out the same optimizations, they are less informative than the metric we have proposed.

Some recent papers on efficiency in statistical parsing have looked at the number of pops from a heap as the relevant measure of efficiency (Caraballo and Charniak, 1998; Charniak, Goldwater, and Johnson, 1998; Blaheta and Charniak, 1999), and have demonstrated techniques for improving the scoring function so that this number is dramatically reduced. This is also a score that cannot be "artificially" reduced through optimization. It may very well be, however, that some significant part of a parser's function is not an operation on a heap. For example, a parser could run a part-of-speech (POS) tagger over the string as a first stage. What is relevant for this first stage are the number of (POS,word) pairs that must be considered by the tagger. Each of these pairs would have a score calculated for them, and would hence be an *event considered*. The events in the second stage may be, for example, edges in the chart. A parser's efficiency score would be the total number of these considered events across all stages.

The principle merits of this metric are that it is general enough to cover different search and pruning techniques (including exhaustive parsing); that it is machine-independent; and that it is, to a certain extent, implementation-independent. The last of these might be what recommends the metric most, insofar as it is not the case for other simple metrics. For example, using time as a metric is perfectly general, and there are ways to normalize for processor differences (see Moore, 2000b). However, unless one is comparing two implementations that are essentially identical in all incidental ways, it is not possible to normalize for certain specifics of the implementation. For example, how probabilities are accessed, upon which processing time is very dependent, can differ from implementation to implementation (see discussion below). Thus, while time may be ideal for highly controlled studies of relatively similar algorithms (as in Moore, 2000a), its applicability for comparing diverse parsers is problematic.

Let us consider a specific example: calculating scores from highly conditioned, interpolated probability distributions. First we will discuss conditional probability models, followed by an illustration of interpolation.

A simple probabilistic context free grammar (PCFG) is a context free grammar with a probability assigned to each rule: the probability of the righthand side of the rule given the lefthand side of the rule. These probabilities can be estimated via their relative frequency in a corpus of trees. For instance, we can assign a probability to the rule S → NP VP by counting the number of occurrences of this rule in the corpus, and dividing by the total number of S nodes in the corpus. We can improve the probability model if we add in more conditioning events beyond the lefthand side of the rule. For example, if we throw in the parent of the lefthand side in the tree within which it appears, we can immediately see a dramatic improvement in the maximum likelihood parse (Johnson, 1998). That is, instead of:

$$P(RHS|LHS) \quad = \quad \frac{P(LHS, RHS)}{P(LHS)}$$

the probability of the rule instance is:

$$P(RHS|LHS, P_{LHS}) \quad = \quad \frac{P(LHS, RHS, P_{LHS})}{P(LHS, P_{LHS})}$$

where P_{LHS} is the parent above the lefthand side of the rule. This additional conditioning event allows us to capture the fact that the distribution of, say, S node expansions underneath VPs is quite different than that of S nodes at the root of the tree. The models that we will be discussing in this paper condition on many such events, somewhere between five and ten. This can lead to sparse data problems, necessitating some kind of smoothing - in these cases, deleted interpolation.

The idea behind deleted interpolation (Jelinek and Mercer, 1980) is simple: mix the empirically observed probability using n conditioning events with lower order models. The mixing coefficients, λ_n, are functions of the frequency of the joint occurrence of the conditioning events, estimated from a held out portion of the corpus. Let e_0 be the event whose probability is to be conditioned, $e_1 \dots e_n$ the n conditioning events used in the model, and $\hat{P}$ the empirically observed conditional probability. Then the following is a recursive definition of the interpolated probability:

$$\begin{aligned} P(e_0|e_1 \dots e_n) &= \lambda_n(e_1 \dots e_n)\hat{P}(e_0|e_1 \dots e_n) + \\ &\quad (1-\lambda_n(e_1 \dots e_n))P(e_0|e_1 \dots e_{n-1}) \end{aligned}$$

This has been shown to be very effective in circumstances where sparse data requires smoothing to avoid assigning a probability of zero to a large number of possible events that happen not to have been observed in the training data with the n conditioning events.

Using such a model[2], the time to calculate a particular conditional probability can be significant. There are a variety of techniques that can be used to speed this up, such as precompilation or caching. These techniques can have a fairly large effect on the time of computation, but they contribute little to a comparison between pruning techniques or issues of search. More generally, optimization and lack of it is something that can obscure algorithm similarities or differences, over and above differences in machine or platform. Researchers whose interest lies in improving parser accuracy might not care to improve the efficiency once it reaches an acceptable level. This should not bar us from trying to compare their techniques with regards to efficiency.

Another such example contrasts our metric with one that measures total heap operations. Depending on the pruning method, it might be possible to evaluate an event's probability and throw it away if it falls below some threshold, rather than pushing it onto the heap. Another option in the same circumstance is to simply push all analyses onto the heap, and let the heap ranking decide if they ever surface again. Both have their respective time trade-offs (the cost of thresholding versus heap operations), and which is chosen is an implementation issue that is orthogonal to the relative search efficiency that we would like to evaluate.

In contrast to time or total heap operations, there is no incidental optimization that allows the parser to avoid calculating scores for analyses. A statistical parser that prunes the search space cannot perform this pruning without scoring events that must be either retained or discarded. A reduction in events considered without a loss of accuracy counts as a novel search or pruning technique, and as such should be explicitly evaluated as a competitor strategy. The basic point that we are making here is that our metric measures that which is central to statistical parsing techniques, and not something that can be incidentally improved.

In the next section, we outline two quite different statistical parsers, and present some results using our new metric.

2 Comparing statistical parsers

To illustrate the utility of this metric for comparing the efficiency of radically different approaches to broad-coverage parsing, we will contrast some results from a two-stage best-first parser (Charniak, 2000) with a single-pass left-to-right, incremental beam-search parser (Roark, 2000). Both of these parsers (which we will refer to, henceforth, as the EC and BR parsers, respectively) score between 85 and 90 percent average precision and recall; both condition the probabilities of events on a large number of contextual parameters in more-or-less the way outlined above; and both use boundary statistics to assign partial structures a figure-of-merit, which is the product of the probability of the structure in its own right and a score for its likelihood of integrating with its surrounding context.

Both of the parsers also use parameterized pruning strategies, which will be described when the parsers are outlined. Results will be presented for each parser at a range of parameter values, to give a sense of the behavior of the parser as more or fewer events are taken into consideration. From this data, we shall be able to see the degree to which the events considered score correlates with time, as well as the convergence in accuracy.

The parsers were trained on sections 2-21 and tested on section 23 of the Penn Wall St. Journal Treebank (Marcus, Santorini, and Marcinkiewicz, 1993), which are the standards in the statistical parsing literature. Accuracy is reported in terms of average labelled precision and recall. Precision is the number of correct constituents divided by the number of

[2]The same points hold for other smoothing methods, such as backing off.

section 23: 2416 sentences of length ≤ 100 Average length: 23.46 words/sentence			
Times past first parse	Avg. Prec/Rec	Events Considered[†]	Time in seconds[†]
21	89.7	212,014	26.7
13	89.6	107,221	14.0
7.5	89.1	48,606	6.7
2.5	86.8	9,621	1.5
2	85.6	6,826	1.1

[†]per sentence

Table 1: Results from the EC parser at different initial parameter values

constituents proposed by the parser. Recall is the number of correct constituents divided by the number of constituents in the actual parse. Labelled precision and recall counts only non-part-of-speech non-terminal constituents. The two numbers are generally quite close, and are averaged to give a single composite score.

2.1 EC parser

The EC parser first prunes the search space by building a chart containing only the most likely edges. Each new edge is assigned a figure-of-merit (FOM) and pushed onto a heap. The FOM is the product of the probability of the constituent given the simple PCFG and the boundary statistics. Edges that are popped from the heap are put into the chart, and standard chart building occurs, with new edges being pushed onto the heap. This process continues until a complete parse is found; hence this is a best-first approach. Of course, the chart building does not necessarily need to stop when the first parse is found; it can continue until some stopping criterion is met. The criterion that was used in the trials that will be reported here is a multiple of the number of edges that were present in the chart when the first parse was found. Thus, if the parameter is 1, the parser stops when the first parse is found; if the parameter is 10, the parser stops when the number of edges in the chart is ten times the number that were in the chart when the first parse was found.

This is the first stage of the parser. The second stage takes all of the parses packed in the chart that are above a certain probability threshold given the PCFG, and assigns a score using the full probability model. To evaluate the probability of each parse, the evaluation proceeds from the top down. Given a particular constituent, it first evaluates the probability of the part-of-speech of the head of that constituent, conditioned on a variety of contextual information from the context. Next, it evaluates the probability of the head itself, given the part-of-speech that was just predicted (plus other information). Finally, it evaluates the probability of the rule expansion, conditioned on, among other things, the POS of the head and the head. It then moves down the tree to evaluate the newly predicted constituents. See Charniak (2000) for more details on the specifics of the parser.

Notice that the events are heterogeneous. One of the key events in the model is the constituent/head relation, which is not an edge. Note also that this two-stage search strategy means that many edges will be considered multiple times, once by the first stage and in every complete parse within which they occur in the second stage, and hence will be counted multiple times by our metric.

The parse with the best score is returned for evaluation in terms of precision and recall. Table 1 shows accuracy and efficiency results when the EC parser is run at various initial parameter values, i.e. the number of times past the first parse the first-stage of the parser continues.

2.2 BR parser

The BR parser proceeds from left-to-right across the string, building analyses top-down in a single pass. While its accuracy is several points below that of the EC parser, it is useful in circumstances requiring incremental processing, e.g. on-line speech recognition, where a multi-stage parser is not an option.

Very briefly, partial analyses are ranked by a figure-of-merit that is the product of their probability (using the full conditional probability model) and a look-ahead probability, which is a measure of the likelihood of the current stack state of an analysis rewriting to the look-ahead word at its left-corner. Partial analyses are popped from the heap, expanded, and pushed back onto the heap. When an analysis is found that extends to the look-ahead word, it is pushed onto a new heap, which collects these "successful" analyses until there are "enough", at which point the look-ahead is moved to the next word in the string, and all of the "unsuc-

section 23: 2416 sentences of length ≤ 100 Average length: 23.46 words/sentence				
Base Beam Factor	Avg. Prec/Rec	Events Considered†	Time in seconds†	Pct. failed
10^{-12}	85.9	265,509	7.6	1.3
10^{-11}	85.7	164,127	4.3	1.7
10^{-10}	85.3	100,439	2.7	2.2
10^{-8}	84.3	36,861	0.9	3.8
10^{-6}	81.8	13,512	0.4	7.1

†per sentence

Table 2: Results from the BR parser at different initial parameter values

cessful" analyses are discarded. This is a beam-search, and the criterion by which it is judged that "enough" analyses have succeeded can be either narrow (i.e. stopping early) or wide (i.e. stopping late). The unpruned parse with the highest probability that successfully covers the entire input string is evaluated for accuracy.

The beam parameter in the trials that will be reported here, is called the base beam factor, and it works as follows. Let β be the base beam factor, and let $\tilde{p}$ be the probability of the highest ranked "successful" parse. Then any analysis whose probability falls below $\alpha\beta\tilde{p}$, where α is the cube of the number of successful analyses, is discarded. The basic idea is that we want the beam to be very wide if there are few analyses that have extended to the current look-ahead word, but relatively narrow if many such analyses have been found. Thus, if $\beta = 10^{-12}$, and 100 analyses have extended to the current look-ahead word, then a candidate analysis must have a probability above $10^{-6}\tilde{p}$ to avoid being pruned. After 1000 candidates, the beam has narrowed to $10^{-3}\tilde{p}$. Table 2 shows accuracy and efficiency results when the BR parser is run at various base beam factors. See Roark (2000) for more details on the specifics of this parser.

The conditional probability model that is used in the BR parser is constrained by the left-to-right nature of the algorithm. Whereas the conditional probability model used in the second stage of the EC parser has access to the full parse trees, and thus can condition the structures with information from either the left or right context, any model used in the BR parser can only use information from the left-context, since that is all that has been built at the moment the probability of a structure is evaluated. For example, a subject NP can be conditioned on the head of the sentence (usually the main verb) in the EC parser, but not in the BR parser, since the head of sentence has yet to be encountered. This accounts for some of the accuracy difference between the two parsers. Also, note that the BR parser can and does fail to find a parse in some percentage of cases, as a consequence of the incremental beam-search. This percentage is reported as well.

3 Discussion

The number of ways in which these two parsers differ is large, and many of these differences make it difficult to compare their relative efficiency. A partial list of these complicating differences is the following:

- Best-first vs. beam search pruning strategy, which impacts the number of events that must be retained
- Two-stage vs. single pass parsing
- Heterogeneous events, within and between parsers
- Different conditional probability models, with different numbers of conditioning events, and slightly different methods of interpolation
- EC parser written in C++; BR parser written in C

In addition, for these runs, the EC parser parallelized the processing by sending each sentence individually off to different processors on the network, whereas the BR parser was run on a single computing server. Since for the EC parser we do not know which sentence went to which

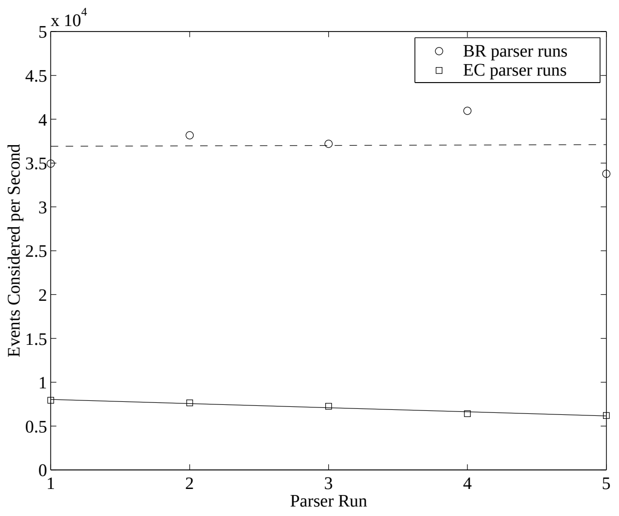


Figure 1: Events Considered per Second for each parser run, with a linear fit

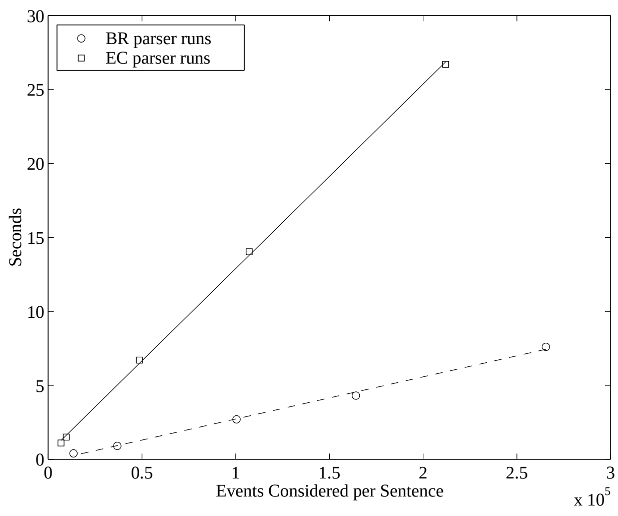


Figure 2: Time vs. Events Considered per Sentence, with a linear fit

processor, nor how fast each individual processor was, time is a particularly poor point of comparison.

In order for our metric to be useful, however, it should be highly correlated with time. Figure 1 shows the number of events considered divided by the total parse time for each of the five runs reported for each parser. While there is some noise between each of the runs, this ratio is relatively constant across the runs, as shown by the linear fit, indicating a very high correlation between the number of events considered and the total time. Figure 2 plots the edges considered versus time per sentence for all of the runs reported in the tables above, and the linear fit for each is drawn as well. As we can see from both plots, number of events considered is a good proxy measure for time in both parsers.

Now the question is how to judge the relative efficiency using this measure. Given that both parsers are parameterized, the number of events considered can be made essentially arbitrarily high or arbitrarily low. We should thus look at the performance of the parsers over a range of parameter values. Figure 3 shows the convergence in accuracy of the models, as more and more events are considered. The improvement in accuracy in the graph is represented as a reduction in parser error, i.e. 100 - average precision/recall. Both of the parsers show a fairly similar pattern of convergence to their respective minimum errors.

Given this information, there are two directions that one can go. The first is to simply take this information at face value and make judgments about the relative efficiency on the basis of these numbers. We may, however, want to take the comparison one step further, and look at how quickly each parser converges to its respective best accuracy, regardless of what that best accuracy is. In a sense, this would focus the evaluation on the search aspects of the parser, apart from the overall quality of the probability model.

Figure 4 plots the percentage of the highest accuracy parse achieved versus the number of events considered. The convergence of the BR parser lies to the right of the convergence of the EC parser, indicating that the EC parser takes fewer edges considered to converge on the best possible accuracy given the model. Notice that both parsers had runs with approximately 100,000 events considered, but that the EC parser is within .1 percent of the best accuracy (basically within noise) at that point, while the BR parser still has a fair amount of improvement to go before reaching the best accuracy. Thus the EC parser needs to consider fewer events to find the best parse.

This is hardly surprising given what we know about the pruning strategies. The first stage of the EC parser uses dynamic programming techniques on the chart to evaluate edges only once. The BR parser, in contrast, must evaluate con-

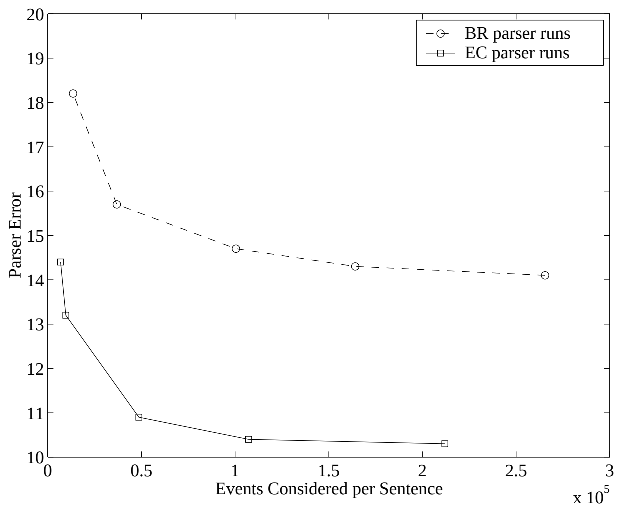


Figure 3: Reduction in parser error as the number of events considered increases

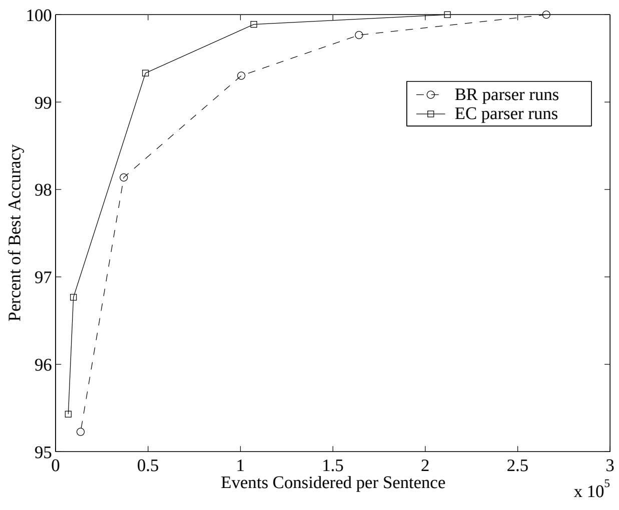


Figure 4: Convergence to highest accuracy parse as number of events considered increase

stituents once for every parse within which they occur. Particularly useless constituents will be thrown out once by the EC parser, but perhaps many times by the BR parser.

This difference in efficiency is tangible, but it is relatively small. What would be problematic in this domain would be orders of magnitude differences, which we don't get here.

4 Conclusion

We have presented in this paper a very general, machine- and implementation-independent metric that can be used to compare the efficiency of quite different statistical parsers. To illustrate its usefulness, we compared the performance of two parsers that follow different strategies in arriving at their parses, and which on the surface would appear to be very difficult to compare with respect to efficiency. Despite this, the two algorithms seem to require a fairly similar number of events considered to squeeze the most accuracy out of their respective models. Furthermore, the decrease in events considered in both cases was accompanied by a more-or-less proportional decrease in time. This data confirmed our intuitions that the two algorithms are roughly similar in terms of efficiency. It also lends support to consideration of this metric as a legitimate, machine and implementation independent measure of statistical parser efficiency.

In practice, the scores on this measure could be reported alongside of the standard PARSEVAL accuracy measures (Black et al., 1991), as an indicator of the amount of work required to arrive at the parse. What is this likely to mean to researchers in high accuracy, broad-coverage statistical parsing? Unlike accuracy measures, whose fluctuations of a few tenths of percent are attended to with interest, such an efficiency score is likely to be attended to only if there is an order of magnitude difference. On the other hand, if two parsers have very similar performance in accuracy, the relative efficiency of one over the other may recommend its use.

When can this metric be used to compare parsers? We would contend that it can be used whenever measures such as precision and recall can be used, i.e. same training and testing corpora. If the parser is working in an entirely different search space, such as with a dependency grammar, or when the training or testing portions of the corpus are different, then it is not clear that such comparisons provide any insight into the relative merits of different parsers. Much of the statistical parsing literature has settled on specific standard training and testing corpora, and in this circumstance, this measure should be useful for evaluation of efficiency.

In conclusion, our efficiency metric has tremendous generality, and is tied to the operation of statistical parsers in a way that recommends its use over time or heap operations as a measure of efficiency.

References

Black, E., S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. 1991. A procedure for quantitatively comparing the syntactic coverage of english grammars. In *DARPA Speech and Natural Language Workshop*, pages 306–311.

Blaheta, D. and E. Charniak. 1999. Automatic compensation for parser figure-of-merit flaws. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 513–518.

Caraballo, S. and E. Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298.

Charniak, E. 1997. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Menlo Park. AAAI Press/MIT Press.

Charniak, E. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 1st Conference of the North American Chapter of the Association for Computational Linguistics.*

Charniak, E., S. Goldwater, and M. Johnson. 1998. Edge-based best-first chart parsing. In *Proceedings of the Sixth Workshop on Very Large Corpora*, pages 127–133.

Collins, M.J. 1997. Three generative, lexicalised models for statistical parsing. In *The Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23.

Collins, M.J. 2000. Discriminative reranking for natural language parsing. In *The Proceedings of the 17th International Conference on Machine Learning.*

Jelinek, F. and R.L. Mercer. 1980. Interpolated estimation of markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*, pages 381–397.

Johnson, M. 1998. PCFG models of linguistic tree representations. *Computational Linguistics*, 24(4):617–636.

Marcus, M.P., B. Santorini, and M.A. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Moore, R. 2000a. Improved left-corner chart parsing for large context-free grammars. In *Proceedings of the Sixth International Workshop on Parsing Technologies*, pages 171–182.

Moore, R. 2000b. Time as a measure of parsing efficiency. In *Proceedings of the COLING-00 Workshop on Efficiency in Large-scale parsing systems.*

Ratnaparkhi, A. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 1–10.

Roark, B. 2000. Probabilistic top-down parsing and language modeling. Submitted.

Some Experiments on Indicators of Parsing Complexity for Lexicalized Grammars

Anoop Sarkar, Fei Xia and Aravind Joshi
Dept. of Computer and Information Science
University of Pennsylvania
200 South 33rd Street,
Philadelphia, PA 19104-6389, USA
{anoop,fxia,joshi}@linc.cis.upenn.edu

Abstract

In this paper, we identify syntactic lexical ambiguity and sentence complexity as factors that contribute to parsing complexity in fully lexicalized grammar formalisms such as Lexicalized Tree Adjoining Grammars. We also report on experiments that explore the effects of these factors on parsing complexity. We discuss how these constraints can be exploited in improving efficiency of parsers for such grammar formalisms.

1 Introduction

The time taken by a parser to produce derivations for input sentences is typically associated with the length of those sentences. The longer the sentence, the more time the parser is expected to take. However, complex algorithms like parsers are typically affected by several factors. A common experience is that parsing algorithms differ in the number of edges inserted into the chart while parsing. In this paper, we explore some of these constraints from the perspective of lexicalized grammars and explore how these constraints might be exploited to improve parser efficiency.

We concentrate on the problem of parsing using *fully* lexicalized grammars by looking at parsers for Lexicalized Tree Adjoining Grammar (LTAG). By a fully lexicalized grammar we mean a grammar in which there are one or more syntactic structures associated with each lexical item. In the case of LTAG each structure is a tree (or, in general, a directed acyclic graph). For each structure there is an explicit structural slot for each of the arguments of the lexical item. The various advantages of defining a lexicalized grammar formalism in this way are discussed in (Joshi and Schabes, 1991).

An example LTAG is shown in Figure 1. To parse the sentence *Ms. Haag plays Elianti* the parser has to combine the trees selected by each word in the sentence by using the operations of substitution and adjunction (the two composition operations in LTAG) producing a valid derivation for the sentence.

Notice that as a consequence of this kind of lexicalized grammatical description there might be several different factors that affect parsing complexity. Each word can select many different trees; for example, the word *plays* in Figure 1 might select several trees for each syntactic context in which it can occur. The verb *plays* can be used in a relative clause, a wh-extraction clause, among others. While grammatical notions of argument structure and syntax can be processed in abstract terms just as in other kinds of formalisms, the crucial difference in LTAG is that all of this information is compiled into a finite set of trees *before* parsing. Each of these separate lexicalized trees is now considered by the parser. This compilation is repeated for other argument structures, e.g. the verb *plays* could also select trees which are intransitive thus increasing the set of lexicalized trees it can select. The set of trees selected by different lexical items is what we term in this paper as *lexical syntactic ambiguity.*

The importance of this compilation into a set of lexicalized trees is that each predicate-argument structure across each syntactic context has its own lexicalized tree. Most grammar formalisms use feature structures to capture the same grammatical and predicate-argument information. In LTAG, this larger set of lexicalized trees directly corresponds to the fact that recursive feature structures are not needed for linguistic description. Feature structures are typically atomic with a few instances of re-entrant features.

Thus, in contrast with LTAG parsing, parsing for formalisms like HPSG or LFG concentrates on efficiently managing the unification of large feature structures and also the packing of ambiguities when these feature structures subsume each other (see (Oepen and Carroll, 2000) and references cited there). We argue in this paper that the result of having compiled out abstract grammatical descriptions into a set of lexicalized trees allows us to predict the number of edges that will be proposed by the parser even before parsing begins. This allows us to explore novel methods of dealing with parsing complexity that are difficult to consider in formalisms that are not fully lexicalized.

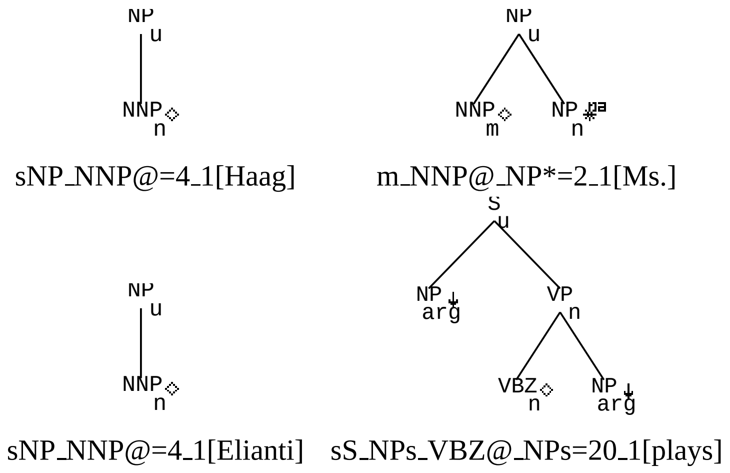


Figure 1: Example lexicalized elementary trees. They are shown in the usual notation: $\diamond$ = *anchor*, $\downarrow$= *substitution node*, $*$ = *footnode*, **na** = *null-adjunction constraint*. These trees can be combined using substitution and adjunction to parse the sentence *Ms. Haag plays Elianti.*

Furthermore, as the sentence length increases, the number of lexicalized trees increase proportionally increasing the attachment ambiguity. Each sentence is composed of several clauses. In a lexicalized grammar, each clause can be seen as headed by a single predicate tree with its arguments and associated adjuncts. We shall see that empirically the number of clauses grow with increasing sentence length only up to a certain point. For sentences greater than a certain length the number of clauses do not keep increasing.

Based on these intuitions we identify the following factors that affect parsing complexity for lexicalized grammars:

Syntactic Lexical Ambiguity The number of trees selected by the words in the sentence being parsed. We show that this is a better indicator of parsing time than sentence length. This is also a predictor of the number of edges that will be proposed by a parser, allowing us to better handle difficult cases *before* parsing.

Sentence Complexity The clausal complexity in the sentences to be parsed. We observe that the number of clauses in a sentence stops growing in proportion to the sentence length after a point. We show that before this point parsing complexity is related to attachment of adjuncts rather than attachment of arguments.

2 LTAG Treebank Grammar

The grammar we used for our experiments was a LTAG Treebank Grammar which was automatically extracted from Sections 02–21 of the Wall Street Journal Penn Treebank II corpus (Marcus et al., 1993). The extraction tool (Xia, 1999) converted the *derived* trees of the Treebank into *derivation* trees in LTAG which represent the attachments of lexicalized elementary trees. There are 6789 tree templates in the grammar with $47,752$ tree nodes. Each word in the corpus selects some set of tree templates. The total number of lexicalized trees is $123,039$. The total number of word types in the lexicon is $44,215$. The average number of trees per word type is 2.78. However, this average is misleading since it does not consider the frequency with which words that select a large number of trees occur in the corpus. In Figure 2 we see that many frequently seen words can select a large number of trees.

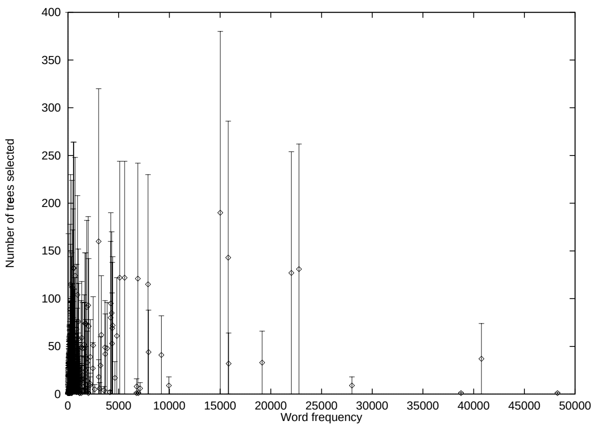


Figure 2: Number of trees selected plotted against words with a particular frequency. (x-axis: words of frequency x; y-axis: number of trees selected, error bars indicate least and most ambiguous word of a particular frequency x)

Another objection that can be raised against a

Treebank grammar which has been automatically extracted is that any parsing results using such a grammar might not be indicative of parsing using a hand-crafted linguistically sophisticated grammar. To address this point (Xia and Palmer, 2000), compares this Treebank grammar with the XTAG grammar (XTAG-Group, 1998), a large-scale hand-crafted LTAG grammar for English. The experiment shows that 82.1% of template tokens in the Treebank grammar matches with a corresponding template in the XTAG grammar; 14.0% are covered by the XTAG grammar but the templates in two grammars look different because the Treebank and the XTAG grammar have adopted different analyses for the corresponding constructions; 1.1% of template tokens in the Treebank grammar are not linguistically sound due to annotation errors in the original Treebank; and the remaining 2.8% are not currently covered by the XTAG grammar. Thus, a total of 96.1% of the structures in the Treebank grammar match up with structures in the XTAG grammar.

3 Syntactic Lexical Ambiguity

In a fully lexicalized grammar such as LTAG the combinations of trees (by substitution and adjunction) can be thought of as *attachments*. It is this perspective that allows us to define the parsing problem in two steps (Joshi and Schabes, 1991):

1. Assigning a set of lexicalized structures to each word in the input sentence.
2. Finding the correct attachments between these structures to get all parses for the sentence.

In this section we will try to find which of these factors determines parsing complexity when finding all parses in an LTAG parser.

To test the performance of LTAG parsing on a realistic corpus using a large grammar (described above) we parsed 2250 sentences from the Wall Street Journal using the lexicalized grammar described in Section 2.[1] All of these sentences were of length 21 words or less. These sentences were taken from the same sections (02-21) of the Treebank from which the original grammar was extracted. This was done to avoid the complication of using default rules for unknown words.

In all of the experiments reported here, the parser produces all parses for each sentence. It produces a shared derivation forest for each sentence which stores, in compact form, all derivations for each sentence.

[1]Some of these results appear in (Sarkar, 2000). In this section we present some additional data on the previous results and also the results of some new experiments that do not appear in the earlier work.

We found that the observed complexity of parsing for LTAG is dominated by factors other than sentence length.[2] Figure 3 shows the time taken in seconds by the parser plotted against sentence length. We see a great deal of variation in timing for the same sentence length, especially for longer sentences.

We wanted to find the relevant variable other than sentence length which would be the right predictor of parsing time complexity. There can be a large variation in syntactic lexical ambiguity which might be a relevant factor in parsing time complexity. To draw this out, in Figure 4 we plotted the number of trees selected by a sentence against the time taken to parse that sentence. By examining this graph we can visually infer that the number of trees selected is a better predictor of increase in parsing complexity than sentence length. We can also compare numerically the two hypotheses by computing the coefficient of determination (R^2) for the two graphs. We get a R^2 value of 0.65 for Figure 3 and a value of 0.82 for Figure 4. Thus, we infer that it is the syntactic lexical ambiguity of the words in the sentence which is the major contributor to parsing time complexity.

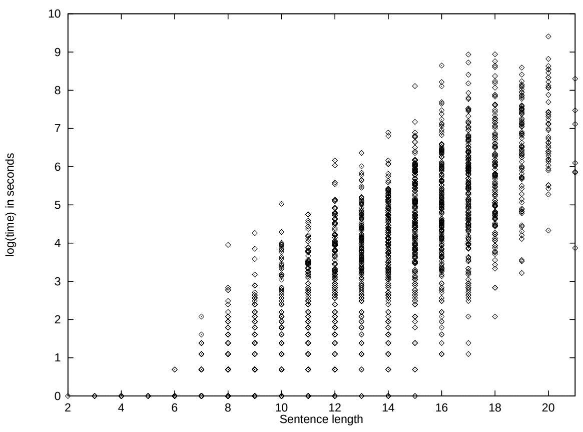


Figure 3: Parse times plotted against sentence length. Coefficient of determination: $R^2 = 0.65$. (x-axis: Sentence length; y-axis: log(time in seconds))

Since we can easily determine the number of trees selected by a sentence before we start parsing, we can use this number to predict the number of edges that will be proposed by a parser when parsing this sentence, allowing us to better handle difficult cases *before* parsing.

[2]Note that the precise number of edges proposed by the parser and other common indicators of complexity can be obtained only while or after parsing. We are interested in *predicting* parsing complexity.

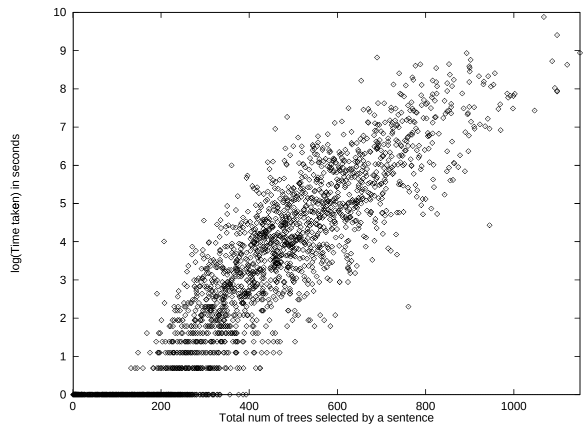


Figure 4: The impact of syntactic lexical ambiguity on parsing times. Log of the time taken to parse a sentence plotted against the total number of trees selected by the sentence. Coefficient of determination: $R^2 = 0.82$. (x-axis: Total number of trees selected by a sentence; y-axis: log(time) in seconds).

We test the above hypothesis further by parsing the same set of sentences as above but this time using an oracle which tells us the correct elementary lexicalized structure for each word in the sentence. This eliminates lexical syntactic ambiguity but does not eliminate attachment ambiguity for the parser. The graph comparing the parsing times is shown in Figure 5. As the comparison shows, the elimination of lexical ambiguity leads to a drastic increase in parsing efficiency. The total time taken to parse all 2250 sentences went from 548K seconds to 31.2 seconds.

Figure 5 shows us that a model which disambiguates syntactic lexical ambiguity can potentially be extremely useful in terms of parsing efficiency. Thus disambiguation of tree assignment or SuperTagging (Srinivas, 1997) of a sentence before parsing it might be a way of improving parsing efficiency. This gives us a way to reduce the parsing complexity for precisely the sentences which were problematic: the ones which selected too many trees. To test whether parsing times are reduced after SuperTagging we conducted an experiment in which the output of an n-best SuperTagger was taken as input to the parser. In our experiment we set n to be 60.[3] The time taken to parse the same set of sentences was again dramatically reduced (the total time taken was 21K seconds). However, the disadvantage of this method was that the coverage of the parser was reduced: 926 sentences (out of the 2250) did not get any parse. This was because some crucial tree was missing in the n-best output. The results are graphed in Figure 6. The total number of derivations for all sentences went down to 1.01e+10 (the original total number was 1.4e+18) indicating (not surprisingly) that some attachment ambiguities persist although the number of trees are reduced. We are experimenting with techniques where the output of the n-best SuperTagger is combined with other pieces of evidence to improve the coverage of the parser while retaining the speedup.

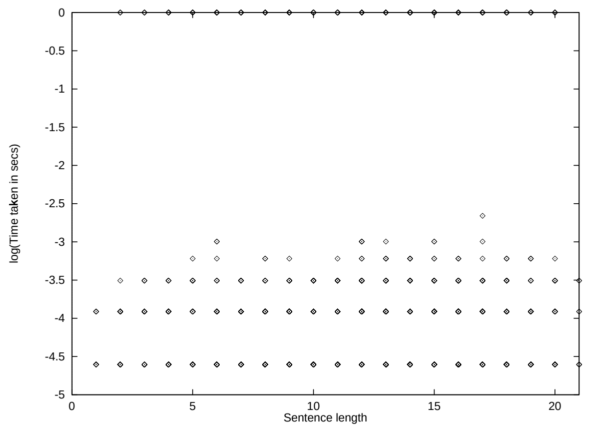


Figure 5: Parse times when the parser gets the correct tree for each word in the sentence (eliminating any syntactic lexical ambiguity). The parsing times for all the 2250 sentences for all lengths never goes above 1 second. (x-axis: Sentence length; y-axis: log(time) in seconds)

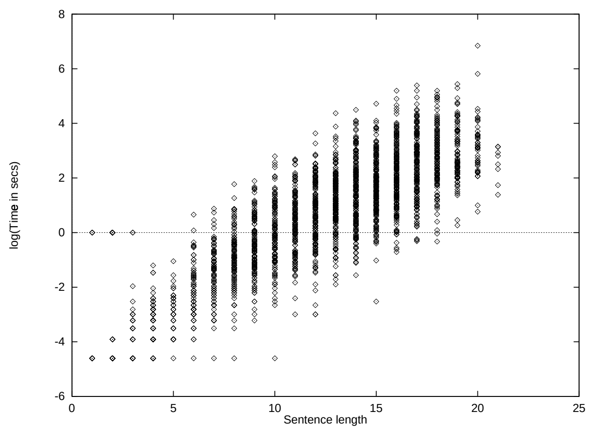


Figure 6: Time taken by the parser after n-best SuperTagging ($n = 60$). (x-axis: Sentence length; y-axis: log(time) in seconds)

4 Sentence Complexity

There are many ways of describing sentence complexity, which are not necessarily independent of

[3](Chen et al., 1999) shows that to get greater than 97% accuracy using SuperTagging the value of n must be quite high ($n > 40$). They use a different set of SuperTags and so we used their result simply to get an approximate estimate of the value of n.

each other. In the context of lexicalized tree-adjoining grammar (and in other lexical frameworks, perhaps with some modifications) the complexity of syntactic and semantic processing is related to the number of predicate-argument structures being computed for a given sentence.

In this section, we explore the possibility of characterizing sentence complexity in terms of the number of clauses which is used as an approximation to the number of predicate-argument structures to be found in a sentence.

The number of clauses of a given sentence in the Penn Treebank is counted using the bracketing tags. The count is computed to be the number of S/SINV/SQ/RRC nodes which have a VP child or a child with -PRD function tag. In principle number of clauses can grow continuously as the sentence length increases. However it is interesting to note that 99.1% of sentences in the Penn Treebank contain 6 or fewer clauses.

Figure 7 shows the average number of clauses plotted against sentence length. For sentences with no more than 50 words, which accounts for 98.2% of the corpus, we see a linear increase in the average number of clauses with respect to sentence length. But from that point on, increasing the sentence length does not lead to a proportional increase in the number of clauses. Thus, empirically, the number of clauses is bounded by a constant. For some very long sentences, the number of clauses actually decreases because these sentences include long but flat coordinated phrases.

Figure 8 shows the standard deviation of the clause number plotted against sentence length. There is an increase in deviation for sentences longer than 50 words. This is due to two reasons: first, quite often, long sentences either have many embedded clauses or are flat with long coordinated phrases; second, the data become sparse as the sentence length grows, resulting in high deviation.[4]

In Figure 9 and Figure 10 we show how parsing time varies as a function of the number of clauses present in the sentence being parsed. The figures are analogous to the earlier graphs relating parsing time with other factors (see Figure 3 and Figure 4). Surprisingly, in both graphs we see that when the number of clauses is small (in this case less than 5), an increase in the number of clauses has no effect on the parsing complexity. Even when the number of clauses is 1 we find the same pattern of time complexity that we have seen in the earlier graphs when we ignored clause complexity. Thus, when the number of clauses is small parsing complexity

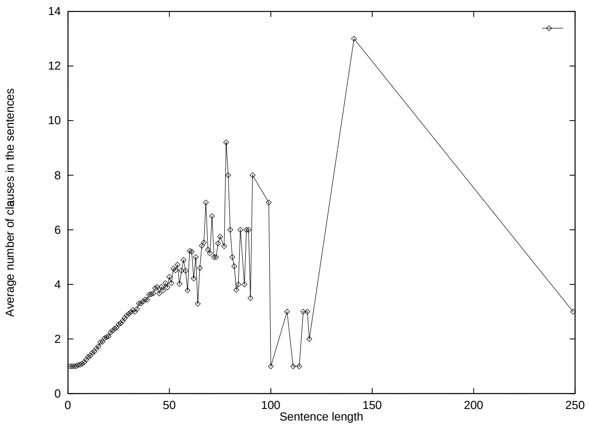


Figure 7: Average number of clause plotted against sentence length

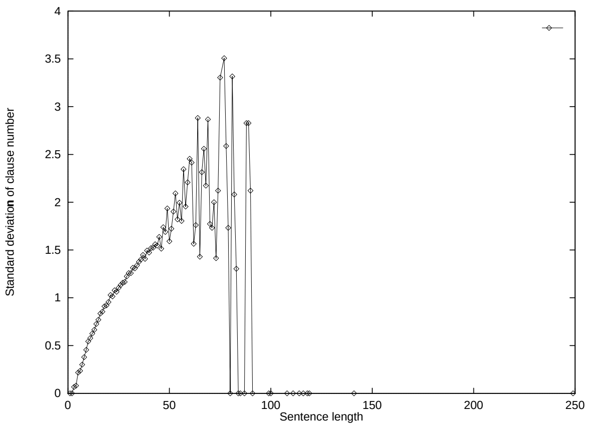


Figure 8: Standard deviation of clause number plotted against sentence length

is related to attachment of adjuncts rather than arguments. It would be interesting to continue increasing the number of clauses and the sentence length and then compare the differences in parsing times.[5]

We have seen that beyond a certain sentence length, the number of clauses do not increase proportionally. We conjecture that a parser can exploit this observed constraint on clause complexity in sentences to improve its efficiency. In a way similar to methods that account for low attachment of adjuncts while parsing, we can introduce constraints on how many clauses a particular node can dominate in a parse. By making the parser sensitive to this measure, we can prune out unlikely derivations previously considered to be plausible by the parser. There is also an independent reason for pursuing this measure of clausal complexity. It can be extended to a notion of syntactic and semantic complexity as they relate to both the representational

[4]For some sentence lengths (e.g., length $= 250$), there is only one sentence with that length in the whole corpus, resulting in zero deviation.

[5]We plan to conduct this experiment and present the results during the workshop.

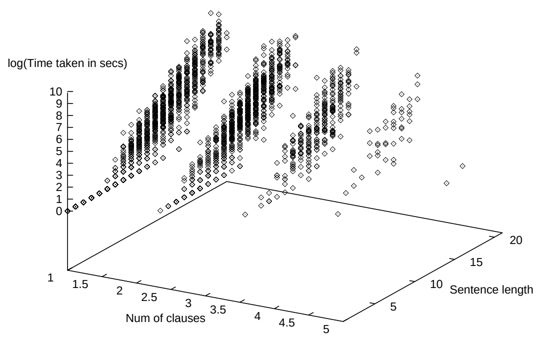


Figure 9: Variation in times for parsing plotted against length of each sentence while identifying the number of clauses.

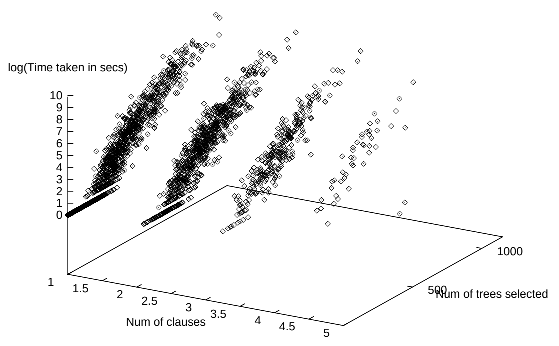


Figure 10: Variation in times for parsing plotted against the number of trees selected by each sentence while identifying the number of clauses.

and processing aspects (Joshi, 2000). The empirical study of clausal complexity described in this section might shed some light on the general issue of syntactic and semantic complexity.

5 Conclusion

In this paper, we identified syntactic lexical ambiguity and sentence complexity as factors that contribute to parsing complexity in fully lexicalized grammars.

We showed that lexical syntactic ambiguity has a strong effect on parsing time and that a model which disambiguates syntactic lexical ambiguity can potentially be extremely useful in terms of parsing efficiency. By assigning each word in the sentence with the correct elementary tree showed that parsing times were reduced by several orders of magnitude (the total time taken to parse 2250 sentences went from 548K seconds to 31.2 seconds).

We conducted an experiment in which the output of an n-best SuperTagger was taken as input to the parser. The time taken to parse the same set of sentences was again dramatically reduced (the total time taken was 21K seconds). The disadvantage of this approach was that 926 out of the original 2250 sentences did not get any parse.

We showed that even as sentence length increases the number of clauses is empirically bounded by a constant. The number of clauses in 99.1% of sentences in the Penn Treebank was bounded by 6. We discussed how this finding affects parsing efficiency and showed that for when the number of clauses is smaller than 4, parsing efficiency is dominated by adjunct attachments rather than argument attachments.

References

John Chen, Srinivas Bangalore, and K. Vijay-Shanker. 1999. New models for improving supertag disambiguation. In *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics*, Bergen, Norway.

A. Joshi and Y. Schabes. 1991. Tree adjoining grammars and lexicalized grammars. In M. Nivat and A. Podelski, editors, *Tree automata and languages*. North-Holland.

Aravind K. Joshi. 2000. Some aspects of syntactic and semantic complexity and underspecification. Talk given at *Syntactic and Semantic Complexity in Natural Language Processing Systems, Workshop at ANLP-NAACL 2000, Seattle*, May.

M. Marcus, B. Santorini, and M. Marcinkiewiecz. 1993. Building a large annotated corpus of english. *Computational Linguistics*, 19(2):313–330.

Stephan Oepen and John Carroll. 2000. Ambiguity packing in constraint-based parsing – practical results. In *Proceedings of the 1st Meeting of the North American ACL, NAACL-2000*, Seattle, Washington, Apr 29 – May 4.

Anoop Sarkar. 2000. Practical experiments in parsing using tree adjoining grammars. In *Proceedings of the Fifth Workshop on Tree Adjoining Grammars*, Paris, France, May 25–27.

The XTAG-Group. 1998. A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS 98-18, University of Pennsylvania.

B. Srinivas. 1997. Performance Evaluation of Supertagging for Partial Parsing. In *Proceedings of Fifth International Workshop on Parsing Technology*, Boston, USA, September.

Fei Xia and Martha Palmer. 2000. Evaluating the Coverage of LTAGs on Annotated Corpora. In *Proceedings of LREC satellite workshop Using Evaluation within HLT Programs: Results and Trends*.

Fei Xia. 1999. Extracting tree adjoining grammars from bracketed corpora. In *Proc. of NLPRS-99*, Beijing, China.

Large Scale Parsing of Czech

Pavel Smrž and **Aleš Horák**
Faculty of Informatics, Masaryk University Brno
Botanická 68a, 602 00 Brno, Czech Republic
E-mail: {smrz,hales}@fi.muni.cz

Abstract

Syntactical analysis of free word order languages poses a big challenge for natural language parsing. In this paper, we describe our approach to feature agreement fulfilment that uses an automatically expanded grammar. We display the insides of the implemented system with its three consecutively produced phases — the core meta-grammar, a generated grammar and an expanded grammar. We present a comparison of parsing with those grammar forms in terms of the parser running time and the number of resulting edges in the chart, and show the need of a shared bank of testing grammars for general parser evaluation.

1 Introduction

Context-Free parsing techniques are well suited to be incorporated into real-world NLP systems for their time efficiency and low memory requirements. However, it is a well-known fact that some natural language phenomena cannot be handled with the context-free grammar (CFG) formalism. Researchers therefore often use the CFG backbone as the core of their grammar formalism and supplement it with context sensitive feature structures (e.g., Pollard and Sag (1994), Neidle (1994)). The mechanism for the evaluation of feature agreement is usually based on unification. The computation can be either interleaved into the parsing process, or it can be postponed until the resulting structure which captures all the ambiguities in syntax has been built (Lavie and Rosé, 2000).

In our approach, we have explored the possibility of shifting the task of feature agreement fulfilment to the earliest phase of parsing process — the CFG backbone. This technique can lead to a combinatorial expansion of the number of rules, however, as we are going to show in this paper, it does not need to cause serious slow-down of the analysis.

In a certain sense, we investigate the interface between phrasal and functional constraints as described in Maxwell III and Kaplan (1991). They compare four different strategies — interleaved pruning, non-interleaved pruning, factored pruning, and factored extraction and see the fundamental asset in the factoring technique. On the other hand, we use a special structure for constraint evaluation. This structure stores all the possible propagated information in one place and allows to solve the functional constraints efficiently at the time of the chart edge closing. Therefore, factoring cannot play such key role in our system.

Maxwell III and Kaplan (1991) further discussed the possibility of translating the functional constraints to the context-free (CF) phrasal constraints and vice versa and noted that "many functional constraints can in principle be converted to phrasal constraints, although converting all such functional constraints is a bad idea, it can be quite advantageous to convert some of them, namely, those constraints that would enable the CF parser to prune the space of constituents". To date, the correct choice of the functional constraints selected for conversion has been explored mostly for English. However, these results cannot simply be applied in morphologically rich languages like Czech, because of the threat of massive expansion of the number of rules. Our preliminary results in answering this question for Czech suggest that converting the functional constraints to CF rules can be valuable for noun phrases, even if the number of rules generated from one original rule can be up to 56 (see below). An open question remains, how to incorporate the process of expansion to other agreement test checking, especially the subject–predicate agreement and verb subcategorization. Here, the cause of problems are the free word order and discontinuity of constituents omnipresent in Czech. Moreover, ellipses (deletions) interfere with the expansion of verb subcategorization constraints and even of the subject–predicate agreement tests (subject can be totally elided in Czech).

2 Description of the System

We bring into play three successive grammar forms. Human experts work with the meta-grammar form, which encompasses high-level generative constructs that reflect the meta-level natural language phenomena like the word order constraints, and enable to describe the language with a maintainable number of rules. The meta-grammar serves as a base for the second grammar form which comes into existence by expanding the constructs. This grammar consists of context-free rules equipped with feature agreement tests and other contextual actions. The last phase of grammar induction lies in the transformation of the tests into standard rules of the expanded grammar with the actions remaining to guarantee the contextual requirements.

Meta-grammar (G1) The meta-grammar consists of global order constraints that safeguard the succession of given terminals, special flags that impose particular restrictions to given non-terminals and terminals on the right hand side and of constructs used to generate combinations of rule elements. The notation of the flags can be illustrated by the following examples:

```
ss -> conj clause
/* budu muset cist -
   I will have to read */
futmod --> VBU VOI VI
/* byl bych byval -
   I would have had */
cpredcondgr ==> VBL VBK VBLL
/* musim se ptat -
   I must ask */
clause ===> VO R VRI
```

The thin short arrow (->) denotes an ordinary CFG transcription. To allow discontinuous constituents, as is needed in Czech syntactic analysis, the long arrow (-->) supplements the right hand side with possible intersegments between each couple of listed elements. The thick long arrow (==>) adds (in addition to filling in the intersegments) the checking of correct enclitics order. This flag is more useful in connection with the order or rhs constructs discussed below. The thick extra-long arrow (===>) provides the completion of the right hand side to form a full clause. It allows the addition of intersegments in the beginning and at the end of the rule, and it also tries to supply the clause with conjunctions, etc.

The global order constraints represent universal simple regulators, which are used to inhibit some combinations of terminals in rules.

```
/* jsem, bych, se -
   am, would, self */
%enclitic   = (VB12, VBK,  R)
/* byl,   cetl, ptal, musel -
   was, read, asked, had to */
%order VBL  = {VL,   VRL,  VOL}
/* byval, cetl, ptal, musel -
   had been, read, asked, had to */
%order VBLL = {VL,   VRL,  VOL}
```

In this example, the %enclitic specifies which terminals should be regarded as enclitics and determines their order in the sentence. The %order constraints guarantee that the terminals VBL and VBLL always go before any of the terminals VL, VRL and VOL.

The main combining constructs in the meta-grammar are order(), rhs() and first(), which are used for generating variants of assortments of given terminals and non-terminals.

```
/* budu se ptat -
   I will ask */
clause ===> order(VBU,R,VRI)
/* ktery ... -
   which ... */
relclause ===> first(relprongr) \
                rhs(clause)
```

The order() construct generates all possible permutations of its components. The first() and rhs() constructs are employed to implant content of all the right hand sides of specified non-terminal to the rule. The rhs(N) construct generates the possible rewritings of the non-terminal N. The resulting terms are then subject to standard constraints and intersegment insertion. In some cases, one needs to force a certain constituent to be the first non-terminal on the right hand side. The construct first(N) ensures that N is firmly tied to the beginning and can neither be preceded by an intersegment nor any other construct. In the above example, the relclause is transformed to CF rules starting with relprongr followed by the right hand sides of the non-terminal clause with possible intersegments filled in.

In the current version, we have added two generative constructs and the possibility to define rule templates to simplify the creation and maintenance of the grammar. The first construct is formed by a set of %list_* expressions, which automatically produce new rules for a list of the given non-terminals either simply concatenated or separated by comma and co-ordinative conjunctions:

```
/* (nesmim) zapomenout udelat -
   to forget to do */
%list_nocoord vi_list
vi_list -> VI

%list_nocoord_case_number_gender modif
/* velky cerveny -
   big red */
modif -> adjp

/* krute a drsne -
   cruelly and roughly */
%list_coord adv_list
adv_list -> ADV

%list_coord_case_number_gender np
/* krasny pes -
   beautiful dog */
np -> left_modif np
...
```

The endings *_case, *_number_gender and *_case_number_gender denote the kinds of agreements between list constituents. The incorporation of this construct has decreased the number of rules by approximately 15%.

A significant portion of the grammar is made up by the verb group rules. Therefore we have been seeking for an instrument that would catch frequent repetitive constructions in verb groups. The obtained addition is the %group keyword illustrated by the following example:

```
%group verb={
    V:head($1,intr)
      add_verb($1),
    VR R:head($1,intr)
         add_verb($1)
         set_R($2)
}

/* ctu - I am reading */
/* ptam se - I am asking */
clause ====> order(group(verb),vi_list)
```

Here, the group verb denotes two sets of non-terminals with the corresponding actions that are then substituted for the expression group(verb) on the right hand side of the clause non-terminal.

Apart from the common generative constructs, the metagrammar comprises feature tagging actions that specify certain local aspects of the denoted (non-)terminal. One of these actions is the specification of the head-dependent relations in the rule — the head() construct:

```
/* prvni clanek - first article */
np -> left_modif np
  head($2,$1)
/* treba - perhaps */
part -> PART
    head(root,$1)
```

In the first rule, head($2,$1) says that (the head of) left_modif depends on (the head of) np on the right hand side. In the second example, head(root,$1) links the PART terminal to the root of the resulting dependency tree. More sophisticated constructs of this kind are the set_local_root() and head_of(), whose usage is demonstrated in the following example:

```
/* ktery ... -
   which ... */
relclause ===> first(relprongr) \
               rhs(clause)
    set_local_root(head_of($2))
```

Here, the heads in rhs(clause) are assigned as specified in the derivation rules for clause. This way we obtain one head of the rhs(clause) part and can link all yet unlinked terms to this head.

The Second Grammar Form (G2) As we have mentioned earlier, several pre-defined grammatical tests and procedures are used in the description of context actions associated with each grammatical rule of the system. We use the following tests:

- grammatical case test for particular words and noun groups

```
noun-gen-group  ->  noun-group \
                      noun-group
   test_genitive($2)
   propagate_all($1)
```

- agreement test of case in prepositional construction

```
prep-group  ->  PREP \
                  noun-group
   agree_case_and_propagate($1,$2)
```

- agreement test of number and gender for relative pronouns

```
ng-with-rel-pron  ->  noun-group \
                   ',' rel-pron-group
   agree_number_gender\
     _and_propagate($1,$3)
```

- agreement test of case, number and gender for noun groups

```
adj-ng  ->  adj-group noun-group
   agree_case_number_gender\
     _and_propagate($1,$2)
```

- test of agreement between subject and predicate
- test of the verb valencies

```
clause  ->  subj-part verb-part
   agree_subj_pred($1,$2)
   test_valency_of($2)
```

The contextual actions propagate_all and agree_*_and_propagate propagate all relevant grammatical information from the non-terminals on the right hand side to the one on the left hand side of the rule.

Expanded Grammar Form (G3) The feature agreement tests can be transformed into the context-free rules. For instance in Czech, similar to other Slavic languages, we have 7 grammatical cases (nominative, genitive, dative, accusative, vocative, locative and instrumental), two numbers (singular and plural) and three genders (masculine, feminine and neuter), in which masculine exists in two forms — animate and inanimate. Thus, e.g., we get 56 possible variants for a full agreement between two constituents.

2.1 Parser

In our work, we have successively tried several different techniques for syntactic analysis. We have tested the top-down and bottom-up variants of the standard chart parser. For more efficient natural language analysis, several researchers have suggested the concept of head-driven parsing (e.g., Kay (1989), van Noord (1997)). Taking advantage of the fact that the head-dependent relations are specified in every rule of our grammar to enable the dependency graph output, the head-driven approach has been successfully adopted in our system. Currently, we are testing the possibility of incorporating the Tomita's GLR parser (Tomita, 1986; Heemels et al., 1991) for the sake of comparing the efficiency of the parsers and the feasibility of implanting a probabilistic control over the parsing process to the parser.

Since the number of rules that we need to work with is fairly big (tens of thousands), we need efficient structures to store the parsing process state. The standard chart parser implementation used in our experiments employs 4 hash structures — one for open edges, one for closed edges, one hash table for the grammar rules (needed in the prediction phase) and one for all edges in the agenda or in the chart (the hash key is made of all the attributes of an edge — the rule, the dot position and the surface range). In the case of a head-driven chart parser, we need two hashes for open edges and also two hashes for closed edges.

The gain of this rather complex structure is the linear dependency of the analysis speed on the number of edges in the resulting chart. Each edge is taken into consideration twice — when it is inserted into the agenda and when it is inserted into the chart. The overall complexity is therefore $2k$, where k is the number of edges in the resulting chart.

The number of chart edges that are involved in the appropriate output derivation structure is related to:

a) the number of words in the input sentence, and

b) the ambiguity rate of the sentence.

The output of the chart parser is presented in the form of a packed shared forest, which is also a standard product of the generalized LR parser. Thus, it enables the parser to run the postprocessing actions on a uniform platform for the different parsers involved.

During the process of design and implementation of our system, we started to distinguish four kinds of contextual actions, tests or constraints:

1. rule-tied actions
2. agreement fulfilment constraints
3. post-processing actions
4. actions based on derivation tree

Rule-tied actions are quite rare and serve only as special counters for rule-based probability estimation or as rule parameterization modifiers. Agreement fulfilment constraints are used in generating the G3 expanded grammar, in G2 they serve as chart pruning actions. In terms of (Maxwell III and Kaplan, 1991), the agreement fulfilment constraints represent the functional constraints, whose processing can be interleaved with that of phrasal constraints. The post-processing actions are not triggered until the chart is already completed. They are used, for instance, in the packed dependency graph generation. On the other hand, there are some actions that do not need to work with the whole chart structure, they are run after the best or n most probable derivation trees are selected. These actions do not prune anything, they may be used, for example, for outputting the verb valencies from the input sentence.

3 Results

In our system, we work with a grammar of the Czech language (Smrž and Horák, 1999), which is being developed in parallel with the parsing mechanism. The grammar in the three forms, as exemplified above, has the following numbers of rules:

G1 meta-grammar – # rules	326
G2 generated grammar – # rules	2919
shift/reduce conflicts	48833
reduce/reduce conflicts	5067
G3 expanded grammar – # rules	10207

As a measure of the ambiguity rate of G2, we display the number of shift/reduce and reduce/reduce conflicts as counted with a standard LR parser generator. These data, together with the number of rules in the grammar, provide basic characteristics of the complexity of analysis.

The comparison of parsing times when using the grammars G3 and G2 without the actions taken into account is summarized in Table 1. We present the time taken for parsing a selected subset of testing sentences — only sentences with more than 50 words were chosen.

The results show that in some cases, which are not so rare in highly inflectional languages, the expanded grammar achieves even less number of edges in the chart than the original grammar. This effect significantly depends on the ambiguity rate of the input text. A question remains, how to exactly characterize the relation between ambiguity in the grammar and in the input.

The fully expanded grammar G3 is only moderately larger than the G2 grammar (about three times the size). The reason lies in the fact that the full expansion takes place mainly in the part of the grammar that describes noun phrases. This part forms only a small amount of the total number of G2 rules. Considering this, it is not surprising that the parse times are not much worse or even better. It also benefits from early pruning by transforming the unification constraints into the CFG. The agreement tests between subject and predicate should also be expanded. Nevertheless, we do not do this, since the position of subject is free, it cannot be described with CF rules without imposing a huge amount of ambiguity to every input sentence.

4 Packed Dependency Graph

Ambiguity is a fundamental property of natural languages. Perhaps the most oppressive case of

Sent #	# of words	G2		G3		# edges G3/G2
		# edges	time	# edges	time	
1100	52	115093	1.95	143044	2.53	124 %
1102	51	84960	1.56	107318	2.02	126 %
1654	51	202678	3.23	361072	6.81	178 %
1672	59	269458	4.08	434430	7.13	161 %
1782	51	212695	3.36	168118	3.05	79 %
2079	66	98363	1.83	223063	3.75	227 %
2300	60	262157	4.03	443022	7.28	169 %
2306	102	739351	12.94	715835	10.95	97 %
2336	103	355749	5.21	565506	8.83	159 %

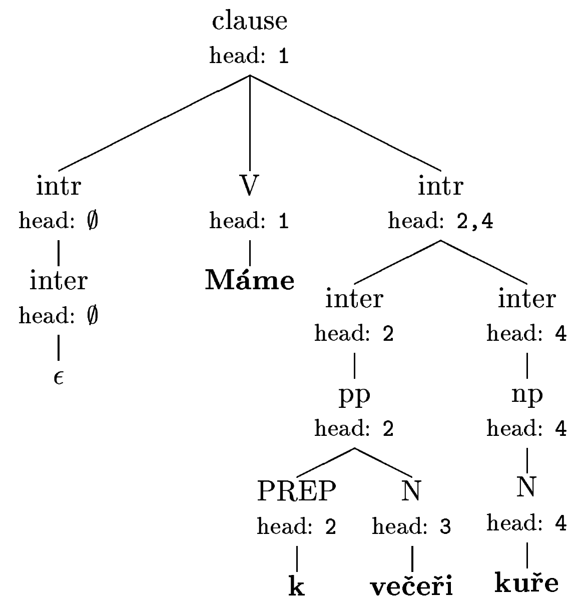
Table 1: Running times for G2 and G3

ambiguity manifests itself on the syntactic level of analysis. In order to face up to the high number of obtained derivation trees, we define a sort order on the output trees that is specified by probabilities computed from appropriate edges in the chart structure. The statistics is also involved in the process of sorting out the edges from the agenda in the order that leads directly to n most probable analyses.

A common approach to acquiring the statistical data for the analysis of syntax employs learning the values from a fully tagged tree-bank training corpus. Building such corpora is a tedious and expensive task and it requires a team cooperation of linguists and computer scientists. At present, the only source of Czech tree-bank data is the Prague Dependency Tree-Bank (PDTB) (Hajič, 1998), which contains dependency analyses of about 20000 Czech sentences.

The linguistic tradition of Czech syntactic analysis is constituted by distinguishing the role of head and dependent and describes the relations between a head and its dependents in terms of semantically motivated dependency relations. In order to be able to exploit the data from PDTB, we have supplemented our grammar with the dependency specification for constituents. Thus, the output of the analysis can be presented in the form of a pure dependency tree. At the same time, we unify classes of derivation trees that correspond to one dependency structure. We then define a canonical form of the derivation to select one representative of the class which is used for assigning the edge probabilities.

The dependency structures for all possible analyses are stored in the form of a packed dependency graph. Every "non-simple" rule (that has more than one term on the right hand side) is extended by a denotation of the head element and its dependents. Thus, the dependency is often given as a relation between non-terminals, which cover several input words. However, the basic units of the dependency representation are particular surface elements (words). To be able to capture the standard dependency relations, we propagate the information about a "local head" from the surface level through all the processed chart edges up to the top. A simplified case that captures only one possible derivation of sentence '`Máme k večeři kuře.`' (*We have a chicken for dinner.*) can be described by the following tree:



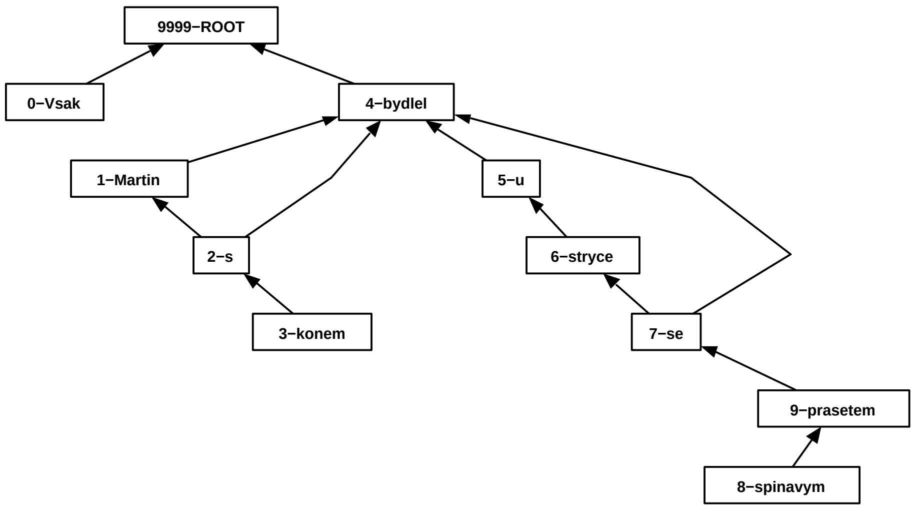


Figure 1: Dependency graph.

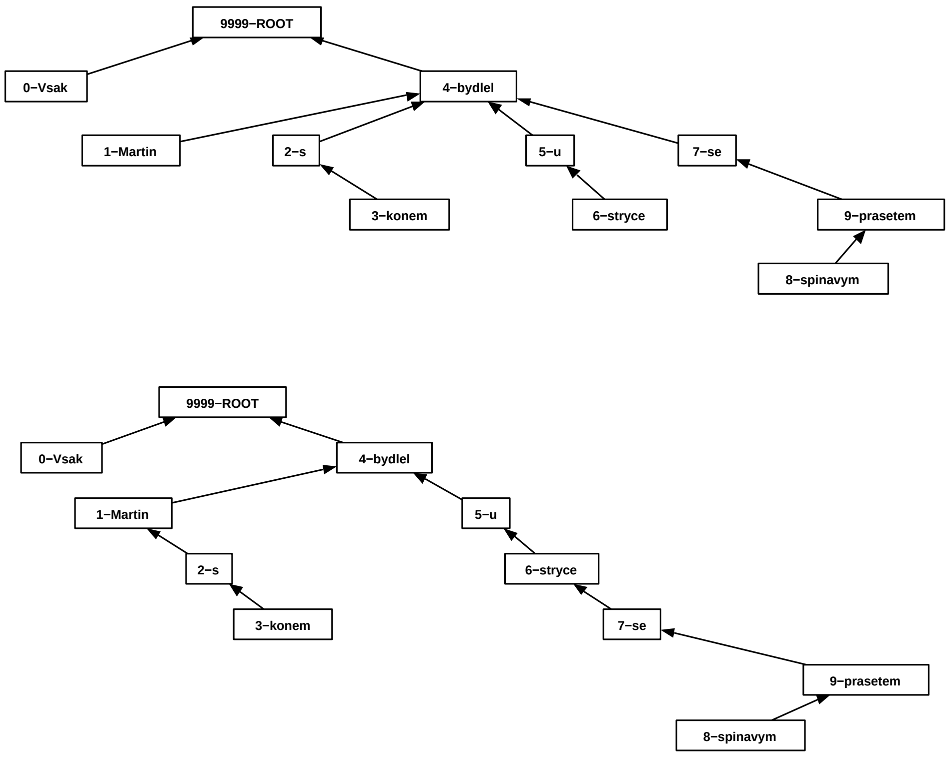


Figure 2: Two of the four possible dependency trees.

During the evaluation of post-processing actions, every head-dependent relation is then recorded as an edge in the graph (without allowing multi-edges). An example of the graph for the sentence '`Však Martin s koněm bydlel u strýce se špinavým prasetem.`' (literally: *However, Martin with a horse lived with his uncle with a dirty pig.*) is depicted in Figure 1. Two examples of unpacked derivation trees that are generated from the graph are illustrated in Figure 2.

The packed dependency graph enables us to recover all the possible standard dependency trees with some additional information gathered during the analysis. The example graph represents two dependency trees only, however, in the case of more complex sentences, especially those with complicated noun phrases, the saving is much higher.

5 Conclusions

In this paper, we have shown that shifting all possible feature agreement computations to the CFG backbone is suitable for free word order languages and it does not need to cause a serious increase in parsing time. We discuss three consecutively produced forms of our grammar and give a comparison of different parser running times on highly ambiguous input.

In the process of parsers evaluation, we lacked the possibility to compare the parsing efficiency on a large number of testing grammars. These grammars cannot be automatically generated, since they should reflect the situation in real-world parsing systems. Future cooperation in NL parsing could therefore lead to the creation of a commonly shared bank of testing grammars with precisely specified ambiguity measures.

References

J. Hajič. 1998. Building a syntactically annotated corpus: The Prague Dependency Treebank. In *Issues of Valency and Meaning*, pages 106–132, Prague. Karolinum.

R. Heemels, A. Nijholt, and K. Sikkel, editors. 1991. *Tomitas Algorithm: Extensions and Applications : Proceedings of the First Twente Workshop on Language Technology*, Enschede. Universiteit Twente.

M. Kay. 1989. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburg.

A. Lavie and P. Rosé, C. 2000. Optimal ambiguity packing in context-free parsers with interleaved unification. In *Proceedings of IWPT'2000*, Trento, Italy.

J. T. Maxwell III and R. M. Kaplan. 1991. The interface between phrasal and functional constraints. In M. Rosner, C. J. Rupp, and R. Johnson, editors, *Proceedings of the Workshop on Constraint Propagation, Linguistic Description, and Computation*, pages 105–120. Instituto Dalle Molle IDSIA, Lugano. Also in Computational Linguistics, Vol. 19, No. 4, 571–590, 1994.

C. Neidle. 1994. Lexical-Functional Grammar (LFG). In R. E. Asher, editor, *Encyclopedia of Language and Linguistics*, volume 3, pages 2147–2153. Pergamon Press, Oxford.

G. Pollard and I. Sag. 1994. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago.

P. Smrž and A. Horák. 1999. Implementation of efficient and portable parser for Czech. In *Proceedings of TSD'99*, pages 105–108, Berlin. Springer-Verlag. Lecture Notes in Artificial Intelligence 1692.

M. Tomita. 1986. *Efficient Parsing for Natural Languages: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Boston, MA.

G. van Noord. 1997. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3).

Demos

Cross-Platform, Cross-Grammar Comparison — Can it be Done?

Ulrich Callmeier and **Stephan Oepen**
Saarland University
Computational Linguistics
{uc | oe}@coli.uni-sb.de

(see 'http://www.coli.uni-sb.de/itsdb/')

Abstract

This software demonstration reviews recent improvements in comparing large-scale unification-based parsing systems, both across different platforms and multiple grammars. Over the past few years significant progress was accomplished in efficient processing with wide-coverage HPSG grammars. A large number of engineering improvements in current systems were achieved through collaboration of multiple research centers and mutual exchange of experience, encoding techniques, algorithms, and pieces of software.

We argue for an approach to grammar and system engineering that makes systematic experimentation and the precise empirical study of system properties a focal point in development. Adapting the profiling metaphor familiar from software engineering to constraint-based grammars and parsers enables developers to maintain an accurate record of system evolution, identify grammar and system deficiencies quickly, and compare to earlier versions, among analytically varied configurations, or between different systems. We demonstrate a suite of integrated software packages facilitating this approach, which are publicly available both separately and together.

The [incr tsdb()] profiling environment (Oepen & Carroll, 2000) integrates empirical assessment and systematic progress evaluation into the development cycle for grammars and processing systems; it enables developers to obtain an accurate snapshot of current system behaviour (a profile) with minimal effort. Profiles can then be analysed and visualized at variable granularity, reflecting various aspects of system competence and performance, and compared to earlier results. Since the [incr tsdb()] package has been integrated with some eight processing platforms by now, it has greatly facilitated cross-fertilization between various research groups and implementations.

PET is a platform for experimentation with processing techniques and the implementation of efficient processors for unification-based grammars (Callmeier, 2000). It synthesizes a range of techniques for efficient processing from earlier systems into a modular C++ implementation, supplying building blocks (such as various unifiers) from which a large number of experimental setups can be configured. A parser built from PET components can be used as a time- and memory-efficient run-time system for grammars developed in the LKB system distributed by CSLI Stanford (Copestake & Flickinger, 2000). In daily grammar development it allows frequent, rapid regression tests.

We emphasize in this demonstration the crucial importance of experimental system comparison, eclectic engineering, and incremental optimization. Only through the careful analysis of a large number of interacting system parameters can one establish reliable points of comparison across different parsers and multiple grammars simultaneously.

References

Callmeier, U. (2000). PET — A platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering, 6 (1) (Special Issue on Efficient Processing with HPSG)*, 99 - 108.

Copestake, A., & Flickinger, D. (2000). An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second Linguistic Resources and Evaluation Conference* (pp. 591 - 600). Athens, Greece.

Oepen, S., & Carroll, J. (2000). Performance profiling for parser engineering. *Natural Language Engineering, 6 (1) (Special Issue on Efficient Processing with HPSG)*, 81 - 97.

Tools for Large-Scale Parser Development

Natural Language Processing Group
Microsoft Research
One Microsoft Way
Redmond WA 98052 USA

1. Introduction

We demonstrate the tool set available to linguistic developers in our NLP lab, with a particular emphasis on the tools for incremental regression testing and creation of regression suites. These tools are currently under use in the daily development of broad-coverage language analysis systems for 7 languages (Chinese, English, French, German, Japanese, Korean and Spanish). The system is modular, with the parsing engine and debugging environments shared by all languages. Linguistic rules are written in a proprietary language (called *G*) whose features are uniquely suited to linguistic tasks (Heidorn, in press). The engine underlying the system, as well as the user interface for linguistic developers, is unicode-enabled thus supporting both European and non-Indo-European languages.

2. Tools for regression testing

The purpose of this class of tools is to build regression suites, which is a collection of what we call *master files*. The master files take the form of stored output trees, and keep a record of the state of development at any point in time.

The linguistic developer builds a set of regression files over the course of grammar development, thus developing annotated corpora. Because the system is intended to cover a broad range of input including ungrammatical input, and because we are very open to letting real text dictate grammar structures rather than theory, we find that annotated structures output by the system are more useful for development than manually tagged corpora.

The standard practice of parser development within our group is schematically shown in Figure 1. As grammar work progresses, developers can run regression tests against the regression suites to examine the consequences of the changes to the grammar. When differences are found, the system gives a color-coded display of the differences (new changes in green, what is in the master file in red, and unchanged part in gray). If the change is an improvement, the developer can choose to update the master file by simply double-clicking on the sentence number on the display, and add the sentences that are newly accommodated to the regression suite. If the change is evaluated as negative, the linguistic developer reworks the rules that caused the regression.

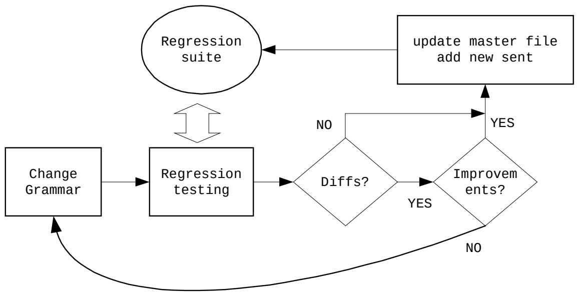


Figure 1. Flow diagram of daily grammar development process

Since we run regression tests many times a day in the grammar development, the processing speed of the systems is a vital issue. Current performance estimates for regression testing are 20 to 30 sentences per second on a 550 MHz Pentium III machine with 512MB RAM across languages (average sentence length = 16.51 words in English, 49.02 chars in Japanese, for example). We also have means to distribute the processing of regression testing onto multiple CPUs: currently, 3 machines with 4 CPUs each (500 MHz, 768MB RAM) regress 27,000 sentences in less than 100 seconds or about 275 sentences per second (English, on the same corpus as above).

3. References

Heidorn, George. In press. Intelligent Writing Assistance. To appear in Robert Dale, Hermann Moisl and Harold Somers (eds.), *Handbook of Natural Language Processing*. Chapter 8.