A Tabulation-Based Parsing Method that Reduces Copying

Gerald Penn and Cosmin Munteanu
Department of Computer Science
University of Toronto
Toronto M5S 3G4, Canada
{gpenn,mcosmin}@cs.toronto.edu

Abstract

This paper presents a new bottom-up chart parsing algorithm for Prolog along with a compilation procedure that reduces the amount of copying at run-time to a constant number (2) per edge. It has applications to unification-based grammars with very large partially ordered categories, in which copying is expensive, and can facilitate the use of more sophisticated indexing strategies for retrieving such categories that may otherwise be overwhelmed by the cost of such copying. It also provides a new perspective on "quick-checking" and related heuristics, which seems to confirm that forcing an early failure (as opposed to seeking an early guarantee of success) is in fact the best approach to use. A preliminary empirical evaluation of its performance is also provided.

1 Introduction

This paper addresses the cost of copying edges in memoization-based, all-paths parsers for phrase-structure grammars. While there have been great advances in probabilistic parsing methods in the last five years, which find one or a few most probable parses for a string relative to some grammar, all-paths parsing is still widely used in grammar development, and as a means of verifying the accuracy of syntactically more precise grammars, given a corpus or test suite.

Most if not all efficient all-paths phrase-structure-based parsers for natural language are chart-based because of the inherent ambiguity that exists in large-scale natural language grammars. Within WAM-based Prolog, memoization can be a fairly costly operation because, in addition to the cost of copying an edge into the memoization table, there is the additional cost of copying an edge out of the table onto the heap in order to be used as a premise in further deductions (phrase structure rule applications). *All* textbook bottom-up Prolog parsers copy edges out: once for every attempt to match an edge to a daughter category, based on a matching end-point node, which is usually the first-argument on which the memoized predicate is indexed. Depending on the grammar and the empirical distribution of matching mother/lexical and daughter descriptions, this number can approach $\mathcal{O}(n)$ copies for an edge added early to the chart, where n is the length of the input to be parsed.

For classical context-free grammars, the category information that must be copied is normally quite small in size. For feature-structure-based grammars and other highly lexicalized grammars with large categories, however, which have become considerably more popular since the advent of the standard parsing algorithms, it becomes quite significant. The ALE system (Carpenter and Penn, 1996) attempts to reduce this by using an algorithm due to Carpenter that traverses the string breadth-first, right-to-left, but matches rule daughters rule depth-first, left-to-right in a failure-driven loop. This eliminates the need for active edges and keeps the sizes of the heap and call stack small. Carpenter's algorithm still copies a candidate edge every time it tries to match it to a daughter description, however, which can approach $\mathcal{O}(n^{b-1})$, where b is the maximum rule branching factor, because of its lack of active edges. [1] The OVIS system (van Noord, 1997) employs *selective memoization*, which tabulates only maximal projections in a head-corner parser — partial projections of a head are still recomputed.

A chart parser with zero copying overhead has yet to be discovered, of course. This paper presents one that reduces this worst case to two copies per non-empty edge, regardless of the length of the input string or when the edge was added to the chart. Since textbook chart parsers require at least two copies per edge as well (assertion and potentially matching the next lexical edge to the left/right), this algorithm always achieves the best-case number of copies attainable by them on non-empty edges. It is thus of some theoretical interest in that it proves that at least a constant bound is attainable within a Prolog setting. It does so by invoking a new kind of grammar transformation, called *EFD-closure*, which ensures that a grammar need not match an empty category to the leftmost daughter of any rule. This transformation is similar to many of the myriad of earlier transformations proposed for exploring the decidability of recognition under various parsing control strategies, but the property it establishes is more conservative than brute-force epsilon elimination for unification-based grammars (Dymetman, 1994). It also still treats empty categories distinctly from non-empty ones, unlike the linking tables proposed for treating leftmost daughters in left-corner parsing (Pereira and Shieber, 1987). Its motivation, the practical consideration of copying overhead, is also rather different.

The algorithm will be presented as an improved version of ALE's parser, although other standard bottom-up parsers can be similarly adapted.

[1] For the same reason, the worst-case time complexity of Carpenter's algorithm is $\mathcal{O}(n^{b+1})$, but a fixed CFG (based on atomic categories) can be converted offline to Chomsky Normal Form in which $b = 2$, thus restoring cubic-time recognition. We are unaware of an equivalent normal form for phrase structure grammars based on typed feature structures.

2 Why Prolog?

Apology! This paper is not an attempt to show that a Prolog-based parser could be as fast as a phrase-structure parser implemented in an imperative programming language such as C. Indeed, if the categories of a grammar are discretely ordered, chart edges can be used for further parsing *in situ*, i.e., with no copying out of the table, in an imperative programming language. Nevertheless, when the categories are partially ordered, as in unification-based grammars, there are certain breadth-first parsing control strategies that require even imperatively implemented parsers to copy edges out of their tables.

What is more important is the tradeoff at stake between efficiency and expressiveness. By improving the performance of Prolog-based parsing, the computational cost of its extra available expressive devices is effectively reduced. The alternative, simple phrase-structure parsing, or extended phrase-structure-based parsing with categories such as typed feature structures, is extremely cumbersome for large-scale grammar design. Even in the handful of instances in which it does seem to have been successful, which includes the recent HPSG English Resource Grammar and a handful of Lexical-Functional Grammars, the results are by no means graceful, not at all modular, and arguably not reusable by anyone except their designers.

Our particular interest in Prolog's expressiveness arises, of course, from an interest in generalized context-free parsing beginning with definite clause grammars (Pereira and Shieber, 1987), as an instance of a logic programming control strategy. The connection between logic programming and parsing is well-known and has also been a very fruitful one for parsing, particularly with respect to the application of logic programming transformations (Stabler, 1993) and constraint logic programming techniques to more recent constraint-based grammatical theories. Relational predicates also make grammars more modular and readable than pure phrase-structure-based grammars.

Commercial Prolog implementations are quite difficult to beat with imperative implementations when it is general logic programming that is required. This is no less true with respect to more recent data structures in lexicalized grammatical theories. A recent comparison (Penn, 2000) between a version of ALE (which is written in Prolog) that reduces typed feature structures to Prolog term encodings, and LiLFeS (Makino et al., 1998), the fastest imperative re-implementation of an ALE-like language, showed that ALE was slightly over 10 times faster on large-scale parses with its HPSG reference grammar than LiLFeS was with a slightly more efficient version of that grammar.

3 Empirical Efficiency

Whether this algorithm will outperform standard Prolog parsers is also largely empirical, because:

1. one of the two copies is kept on the heap itself and not released until the end of the parse. For large parses over large data structures, that can increase the size of the heap significantly, and will result in a greater number of cache misses and page swaps.
2. the new algorithm also requires an offline partial evaluation of the grammar rules that increases the number of rules that must be iterated through at run-time during depth-first closure. This can result in redundant operations being performed among rules and their partially evaluated instances to match daughter categories, unless those rules and their partial evaluations are folded together with local disjunctions to share as much compiled code as possible.

A preliminary empirical evaluation is presented in Section 8.

Oepen and Carroll (2000), by far the most comprehensive attempt to profile and optimize the performance of feature-structure-based grammars, also found copying to be a significant issue in their imperative implementations of several HPSG parsers — to the extent that it even warranted recomputing unifications in places, and modifying the manner in which active edges are used in their fastest attempt (called *hyper-active parsing*). The results of the present study can only cautiously be compared to theirs so far, because of our lack of access to the successive stages of their implementations and the lack of a common grammar ported to all of the systems involved. Some parallels can be drawn, however, particularly with respect to the utility of indexing and the maintenance of active edges, which suggest that the algorithm presented below makes Prolog behave in a more "C-like" manner on parsing tasks.

4 Theoretical Benefits

The principal benefits of this algorithm are that:

1. it reduces copying, as mentioned above.
2. it does not suffer from a problem that textbook algorithms suffer from when running under non-ISO-compatible Prologs (which is to say most of them). In such Prologs, asserted empty category edges that can match leftmost daughter descriptions of rules are not able to combine with the outputs of those rules.
3. keeping a copy of the chart on the heap allows more sophisticated indexing strategies to apply to memoized categories that would otherwise be overwhelmed by the cost of copying an edge before matching it against an index.

Indexing is also briefly considered in Section 8. Indexing is not the same thing as filtering (Torisawa and Tsuji, 1995), which extracts an approximation grammar to parse with first, in order to increase the likelihood of early unification failure. If the filter parse succeeds, the system then proceeds to perform the entire unification operation, as if the approximation had never been applied. Malouf et al. (2000) cite an improvement of 35–45% using a "quick-check" algorithm that they appear to believe finds the optimal selection of n feature paths for quick-checking. It is in fact only a greedy approximation — the optimization problem is exponential in the number of feature paths used for the check. Penn (1999) cites an improvement of 15-40% simply by re-ordering the sister features of only two types in the signature of the ALE HPSG grammar during normal unification.

True indexing *re-orders* required operations without repeating them. Penn and Popescu (1997) build an automaton-based index for surface realization with large lexica, and suggest an extension to statistically trained decision trees. Ninomiya et al. (2002) take a more computationally brute-force approach to index very large databases of feature structures for some kind of information retrieval application. Neither of these is suitable for indexing chart edges during parsing, because the edges are discarded after every sentence, before the expense of building the index can be satisfactorily amortized. There is a fair amount of relevant work in the database and programming language communities, but many of the results are negative (Graf, 1996) — very little time can be spent on constructing the index.

A moment's thought reveals that the very notion of an active edge, tabulating the well-formed prefixes of rule right-hand-sides, presumes that copying is not a significant enough issue to merit the overhead of more specialized indexing. While the present paper proceeds from Carpenter's algorithm, in which no active edges are used, it will become clear from our evaluation that active edges or their equivalent within a more sophisticated indexing strategy are an issue that should be re-investigated now that the cost of copying can provably be reduced in Prolog.

5 The Algorithm

In this section, it will be assumed that the phrase-structure grammar to be parsed with obeys the following property:

Definition 1 *An (extended) context-free grammar, G, is* empty-first-daughter-closed *(*EFD-closed*) iff, for every derivation $N \Rightarrow^* w$, there is a derivation of $N \Rightarrow^* w$ in which no left-corner of any non-preterminal subtree is ϵ.*

In other words, no string relies on deriving the empty string from a leftmost daughter for its membership in the language of the grammar. The next section will show how to transform any phrase-structure grammar into an EFD-closed grammar.

The new parsing algorithm, like Carpenter's algorithm, proceeds breadth-first, right-to-left through the string, at each step applying the grammar rules depth-first, matching daughter categories left-to-right. The first step is then to reverse the input string, and compute its length (performed by `reverse_count/5`) and initialize the chart:

```
rec(Ws,FS) :-
  retractall(edge(_,_,_)),
  reverse_count(Ws,[],WsRev,0,Length),
  CLength is Length - 1,
  functor(Chart,chart,CLength),
  build(WsRev,Length,Chart),
  edge(0,Length,FS).
```

Two copies of the chart are used in this presentation. One is represented by a term `chart(E1,...,EL)`, where the i^{th} argument holds the list of edges whose left node is i. Edges at the beginning of the chart (left node 0) do not need to be stored in this copy, nor do edges beginning at the end of the chart (specifically, empty categories with left node and right node `Length`). This will be called the *term copy* of the chart. The other copy is kept in a dynamic predicate, `edge/3`, as a textbook Prolog chart parser would. This will be called the *asserted copy* of the chart.

Neither copy of the chart stores empty categories. These are assumed to be available in a separate predicate, `empty_cat/1`. Since the grammar is EFD-closed, no grammar rule can produce a new empty category. Lexical items are assumed to be available in the predicate `lex/2`.

The predicate, `build/3`, actually builds the chart:

```
build([W|Ws],R,Chart):-
  RMinus1 is R - 1,
  (lex(W,FS),
   add_edge(RMinus1,R,FS,Chart)
  ; ( RMinus1 =:= 0 -> true
    ; rebuild_edges(RMinus1,Edges),
      arg(RMinus1,Chart,Edges),
      build(Ws,RMinus1,Chart)
    )
  ).
build([],_,_).
```

The precondition upon each call to `build(Ws,R,Chart)` is that `Chart` contains the complete term copy of the non-loop edges of the parsing chart from node `R` to the end, while `Ws` contains the (reversed) input string from node `R` to the beginning. Each pass through the first clause of `build/3` then decrements `Right`, and seeds the chart with every category for the lexical item that spans from `R-1` to `R`. The predicate, `add_edge/4` actually adds the lexical edge to the asserted copy of the chart, and then closes the chart depth-first under rule applications in a failure-driven loop. When it has finished, if `Ws` is not empty (`RMinus1` is not 0), then `build/3` retracts all of the new edges from the asserted copy of the chart (with `rebuild_edges/2`, described below) and adds them to the `R-1`st argument of the term copy before continuing to the next word.

`add_edge/4` matches non-leftmost daughter descriptions from either the term copy of the chart, thus eliminating the need for additional copying of non-empty edges, or from `empty_cat/1`. Whenever it *adds* an edge, however, it adds it to the asserted copy of the chart. This is necessary because `add_edge/4` works in a failure-driven loop, and any edges added to the term copy of the chart would be removed during backtracking:

```
add_edge(Left,Right,FS,Chart):-
  assert(edge(Left,Right,FS)),
  rule(FS,Left,Right,Chart).

rule(FS,L,R,Chart) :-
  (Mother ===> [FS|DtrsRest]), % PS rule
  match_rest(DtrsRest,R,Chart,Mother,L).

match_rest([],R,Chart,Mother,L) :-
  % all Dtrs matched
  add_edge(L,R,Mother,Chart).
match_rest([Dtr|Dtrs],R,Chart,Mother,L) :-
  arg(R,Chart,Edges),
  member(edge(Dtr,NewR),Edges),
  match_rest(Dtrs,NewR,Chart,Mother,L)
 ; empty_cat(Dtr),
   match_rest(Dtrs,R,Chart,Mother,L).
```

Note that we never need to be concerned with updating the term copy of the chart during the operation of add_edge/4 because EFD-closure guarantees that all non-leftmost daughters must have left nodes strictly greater than the Left passed as the first argument to add_edge/4.

Moving new edges from the asserted copy to the term copy is straightforwardly achieved by rebuild_edges/2:

```
rebuild_edges(Left,Edges) :-
  retract(edge(Left,R,FS))
  -> Edges = [edge(FS,R)|EdgesRest],
     rebuild_edges(Left,EdgesRest)
   ; Edges = [].
```

The two copies required by this algorithm are thus: 1) copying a new edge to the asserted copy of the chart by add_edge/4, and 2) copying new edges from the asserted copy of the chart to the term copy of the chart by rebuild_edges/2. The asserted copy is only being used to protect the term copy from being unwound by backtracking.

Asymptotically, this parsing algorithm has the same complexity as Carpenter's algorithm — only its memory consumption and copying behavior are different.

6 EFD-closure

To convert an (extended) context-free grammar to one in which EFD-closure holds, we must partially evaluate those rules for which empty categories could be the first daughter over the available empty categories. If all daughters can be empty categories in some rule, then that rule may create new empty categories, over which rules must be partially evaluated again, and so on. The closure algorithm is presented in Figure 1 in pseudo-code and assumes the existence of six auxiliary lists:

- Es — a list of empty categories over which partial evaluation is to occur,
- Rs — a list of rules to be used in partial evaluation,
- NEs — new empty categories, created by partial evaluation (when all daughters have matched empty categories),
- NRs — new rules, created by partial evaluation (consisting of a rule to the leftmost daughter of which an empty category has applied, with only its non-leftmost daughters remaining),
- EAs — an accumulator of empty categories already partially evaluated once on Rs, and
- RAs — an accumulator of rules already used in partial evaluation once on Es.

```
Initialize Es to empty cats of grammar;
initialize Rs to rules of input grammar;
initialize the other four lists to [];

loop:
while Es =/= [] do
 for each E in Es do
 for each R in Rs do
  unify E with the leftmost unmatched
    category description of R;
  if it does not match, continue;
  if the leftmost category was rightmost
    (unary rule),
   then add the new empty category to NEs
  otherwise, add the new rule (with leftmost
    category marked as matched) to NRs;
 od
 od;
EAs := append(Es,EAs); Rs := append(Rs,RAs);
RAs := []; Es := NEs; NEs := [];
od;
if NRs = [],
 then end: EAs are the closed empty cats,
          Rs are the closed rules
 else
   Es := EAs; EAs := []; RAs := Rs;
   Rs := NRs; NRs := []
   go to loop
```

Figure 1: The offline EFD-closure algorithm.

Each pass through the while-loop attempts to match the empty categories in Es against the leftmost daughter description of every rule in Rs. If new empty categories are created in the process (because some rule in Rs is unary and its daughter matches), they are also attempted — EAs holds the others until they are done. Every time a rule's leftmost daughter matches an empty category, this effectively creates a new rule consisting only of the non-leftmost daughters of the old rule. In a unification-based setting, these non-leftmost daughters could also have some of their variables instantiated to information from the matching empty category.

If the while-loop terminates (see the next section), then the rules of `Rs` are stored in an accumulator, `RAs`, until the new rules, `NRs`, have had a chance to match their leftmost daughters against all of the empty categories that `Rs` has. Partial evaluation with `NRs` may create new empty categories that `Rs` have never seen and therefore must be applied to. This is taken care of within the while-loop when `RAs` are added back to `Rs` for second and subsequent passes through the loop.

7 Termination Properties

As with all bottom-up parsers, the parsing algorithm itself may not terminate because of unary phrase structure rules and/or empty categories. In addition, offline EFD-closure may not terminate when infinitely many new empty categories can be produced by the production rules.

We say that an extended context-free grammar, by which classical CFGs as well as unification-based phrase-structure grammars are implied, is *ϵ-offline-parseable* (ϵ-OP) iff the empty string is not infinitely ambiguous in the grammar. Every ϵ-OP grammar can be converted to a weakly equivalent grammar which has the EFD-closure property. The proof of this statement, which establishes the correctness of the algorithm, is omitted for brevity.

EFD-closure bears some resemblance in its intentions to Greibach Normal Form, but: (1) it is far more conservative in the number of extra rules it must create; (2) it is linked directly to the derivable empty categories of the grammar, whereas GNF conversion proceeds from an already ϵ-eliminated grammar (EFD-closure of any ϵ-free grammar, in fact, is the grammar itself); (3) GNF is rather more difficult to define in the case of unification-based grammars than with classical CFGs, and in the one generalization we are aware of (Dymetman, 1992), EFD-closure is actually not guaranteed by it. Dymetman's generalization, furthermore, only works for classically offline-parseable grammars.

In the case of non-ϵ-OP grammars, a standard bottom-up parser without EFD-closure would not terminate at run-time either. Our new algorithm is thus neither better nor worse than a textbook bottom-up parser with respect to termination. A remaining topic for consideration is the adaptation of this method to strategies with better termination properties than the pure bottom-up strategy.

8 Empirical Evaluation

The details of how to integrate an indexing strategy for unification-based grammars into the EFD-based parsing algorithm are too numerous to present here, but a few empirical observations can be made. First, EFD-based parsing is faster than Carpenter's algorithm even with atomic, CFG-like categories, where the cost of copying is at a minimum, even with no indexing. We defined several sizes of CFG by extracting local trees from successively increasing portions of the Penn Treebank II, as shown in Table 1, and

WSJ directories	Number of WSJ files	Lexicon size	Number of Rules
00	4	131	77
00	5	188	124
00	6	274	168
00	8	456	282
00	10	756	473
00	15	1167	736
00	20	1880	1151
00	25	2129	1263
00	30	2335	1369
00	35	2627	1589
00	40	3781	2170
00	50	5645	3196
00–01	100	8948	5246
00–01	129	11242	6853
00–02	200	13164	7984
00–02	250	14730	9008
00–03	300	17555	10834
00–03	350	18861	11750
00–04	400	20359	12696
00–05	481	20037	13159
00–07	700	27404	17682
00–09	901	32422	20999

Table 1: The grammars extracted from the Wall Street Journal directories of the PTB II.

then computed the average time to parse a corpus of sentences (5 times each) drawn from the initial section. All of the parsers were written in SICStus Prolog. These average times are shown in Figure 2 as a

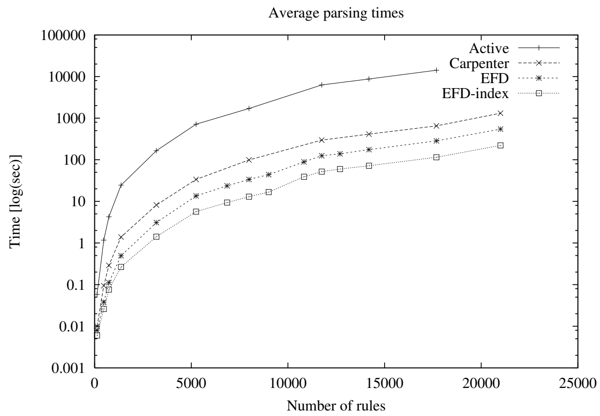

Figure 2: Parsing times for simple CFGs.

function of the number of rules. Storing active edges is always the worst option, followed by Carpenter's algorithm, followed by the EFD-based algorithm. In this atomic case, indexing simply takes on the form of a hash by phrase structure category. This can be implemented on top of EFD because the overhead of copying has been reduced. This fourth option is the fastest by a factor of approximately 2.18 on average over EFD without indexing.

One may also refer to Table 2, in which the number of successful and failed unifications (matches) was counted over the test suite for each rule set. Asymptotically, the success rate does not decrease by very much from rule set to rule set. There are so many more failures early on, however, that the sheer quantity of failed unifications makes it more important to dispense with these quickly.

Number of rules	Successful unifications	Failed unifications	Success rate (%)
124	104	1,766	5.56
473	968	51,216	1.85
736	2,904	189,528	1.51
1369	7,152	714,202	0.99
3196	25,416	3,574,138	0.71
5246	78,414	14,644,615	0.53
6853	133,205	30,743,123	0.43
7984	158,352	40,479,293	0.39
9008	195,382	56,998,866	0.34
10834	357,319	119,808,018	0.30
11750	441,332	151,226,016	0.29
12696	479,612	171,137,168	0.28
14193	655,403	250,918,711	0.26
17682	911,480	387,453,422	0.23
20999	1,863,523	847,204,674	0.21

Table 2: Successful unification rate for the (non-indexing) EFD parser.

Of the grammars to which we have access that use larger categories, this ranking of parsing algorithms is generally preserved, although we have found no correlation between category size and the factor of improvement. John Carroll's Prolog port of the Alvey grammar of English (Figure 3), for example, is EFD-closed, but the improvement of EFD over Carpenter's algorithm is much smaller, presumably because there are so few edges when compared to the CFGs extracted from the Penn Treebank. EFD-index is also slower than EFD without indexing because of our poor choice of index for that grammar. With subsumption testing (Figure 4), the active edge algorithm and Carpenter's algorithm are at an even greater disadvantage because edges must be copied to be compared for subsumption. On a pre-release version of MERGE (Figure 5),[2] a modification of the English Resource Grammar that uses more macros and fewer types, the sheer size of the categories combined with a scarcity of edges seems to cost EFD due to the loss of locality of reference, although that loss is more than compensated for by indexing.

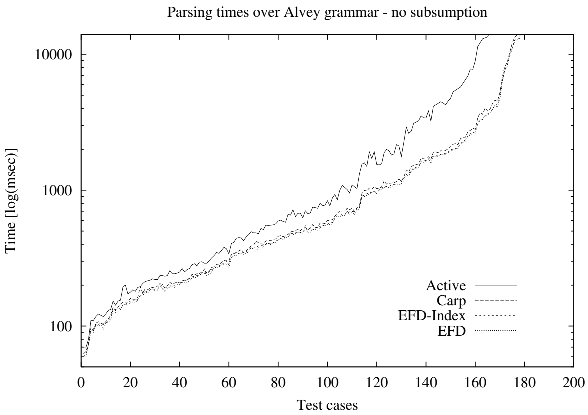

Figure 3: Alvey grammar with no subsumption.

9 Conclusion

This paper has presented a bottom-up parsing algorithm for Prolog that reduces the copying of edges

[2]We are indebted to Kordula DeKuthy and Detmar Meurers of Ohio State University, for making this pre-release version available to us.

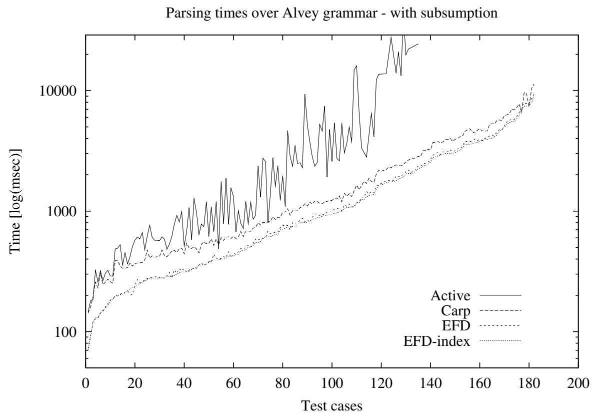

Figure 4: Alvey grammar with subsumption testing.

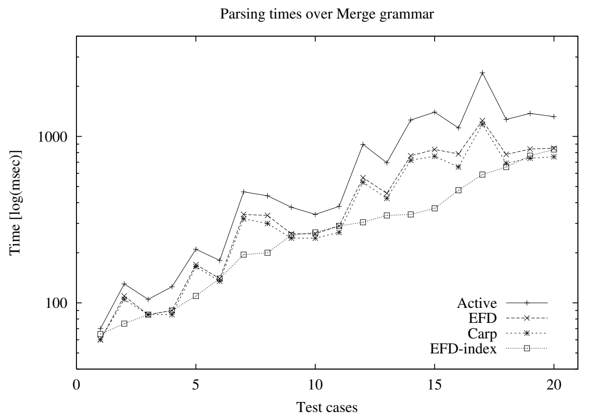

Figure 5: MERGE on the CSLI test-set.

to a constant number of two per non-empty edge. Its termination properties and asymptotic time complexity are the same as those of Carpenter's algorithm, but in practice it performs better. Further optimizations can be incorporated by compiling rules in a way that localizes the disjunctions that are implicit in the creation of extra rules in the compile-time EFD-closure step, and by integrating automaton- or decision-tree-based indexing with this algorithm. With copying now being unnecessary for matching a daughter category description, these two areas should result in a substantial improvement to parse times for highly lexicalized grammars. The adaptation of this algorithm to active edges, other control strategies, and to scheduling concerns such as finding the first parse as quickly as possible remain interesting areas of further extension.

Apart from this empirical issue, this algorithm is of theoretical interest in that it proves that a constant number of edge copies can be attained by an all-paths parser, even in the presence of partially ordered categories.

References

B. Carpenter and G. Penn. 1996. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technologies*, pages 145–168. Kluwer.

M. Dymetman. 1992. A generalized greibach normal form for definite clause grammars. In *Proceedings of the International Conference on Computational Linguistics*.

M. Dymetman. 1994. A simple transformation for offline-parsable gramamrs and its termination properties. In *Proceedings of the International Conference on Computational Linguistics*.

P. Graf. 1996. *Term Indexing*. Springer Verlag.

T. Makino, K. Torisawa, and J. Tsuji. 1998. LiLFeS — practical unification-based programming system for typed feature structures. In *Proceedings of COLING/ACL-98*, volume 2, pages 807–811.

R. Malouf, J. Carroll, and A. Copestake. 2000. Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1):29–46.

T. Ninomiya, T. Makino, and J. Tsuji. 2002. An indexing scheme for typed feature structures. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING-02)*.

S. Oepen and J. Carroll. 2000. Parser engineering and performance profiling. *Natural Language Engineering*.

G. Penn and O. Popescu. 1997. Head-driven generation and indexing in ALE. In *Proceedings of the ENVGRAM workshop; ACL/EACL-97*.

G. Penn. 1999. Optimising don't-care non-determinism with statistical information. Technical Report 140, Sonderforschungsbereich 340, Tübingen.

G. Penn. 2000. *The Algebraic Structure of Attributed Type Signatures*. Ph.D. thesis, Carnegie Mellon University.

F. C. N. Pereira and S. M. Shieber. 1987. *Prolog and Natural-Language Analysis*, volume 10 of *CSLI Lecture Notes*. University of Chicago Press.

E. Stabler. 1993. *The Logical Approach to Syntax: Foundations, Specifications, and implementations of Theories of Government and Binding*. MIT Press.

K. Torisawa and J. Tsuji. 1995. Compiling HPSG-style grammar to object-oriented language. In *Proceedings of NLPRS-1995*, pages 568–573.

G. van Noord. 1997. An efficient implementation of the head-corner parser. *Computational Linguistics*.