Improved Left-Corner Chart Parsing for Large Context-Free Grammars*

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

Abstract

We develop an improved form of left-corner chart parsing for large context-free grammars, introducing improvements that result in significant speed-ups compared to previously-known variants of left-corner parsing. We also compare our method to several other major parsing approaches, and find that our improved left-corner parsing method outperforms each of these across a range of grammars. Finally, we also describe a new technique for minimizing the extra information needed to efficiently recover parses from the data structures built in the course of parsing.

1 Introduction

Parsing algorithms for contex-free grammars (CFGs) are generally recognized as the backbone of virtually all approaches to parsing natural-language. Even in systems that use a grammar formalism more complex than CFGs (e.g., unification grammar), the parsing method is usually an extension of one of the well-known CFG parsing algorithms. Moreover, recent developments have once again made direct parsing of CFGs more relevant to natural-language processing, including the recent explosion of interest in parsing with stochastic CFGs or related formalisms, and the fact that commercial speech recognition systems are now available (from Nuance Communications and Microsoft) that accept CFGs as language models for constraining recognition.

These applications of context-free parsing share the common trait that the grammars involved can be expected to be very large. A "treebank grammar" extracted from the sections of the Penn Treebank commonly used for training stochastic parsers contains over 15,000 rules, and we also have a CFG containing over 24,000 rules, compiled from a task-specific unification grammar for use as a speech-recognition language model. Grammars such as these stress established approaches to context-free parsing in ways and to an extent not encountered with smaller grammars.

In this work we develop an improved form of left-corner chart parsing for large context-free grammars. We introduce improvements that result in speed-ups averaging 38% or more compared to previously-known variants of left-corner parsing. We also compare our method to several other major parsing approaches: Cocke-Kasami-Younger (CKY), Earley/Graham-Harrison-Ruzzo (E/GHR), and generalized LR (GLR) parsing. Our improved left-corner parsing method outperforms each of these by an average of at least 50%. Finally, we also describe a new technique for minimizing the extra information needed to efficiently recover parses from the data structures built in the course of parsing.

*Revised version of paper appearing in *Proceedings of the Sixth International Workshop on Parsing Technologies, IWPT 2000*. Revised 23 March 2000.

2 Evaluating Parsing Algorithms

In this work we are interested in algorithms for finding all possible parses for a given input. We measure the efficiency of the algorithms in building a complete chart (or comparable structure) for the input, where the chart includes information sufficient to recover every parse without additional searching.[1] We take CPU time to be the primary measure of performance. Implementation-independent measures, such as number of chart edges generated, are sometimes preferred in order to factor out the effects of different platforms and implementation methods, but only time measurement provides a practical way of evaluating some algorithmic details. For example, one of our major improvements to left-corner parsing simply transposes the order of performing two independent filtering checks, resulting in speed ups of up to 67%, while producing exactly the same chart edges as the previous method. To ensure comparability of time measurements, we have re-implemented all the algorithms considered, in Perl 5,[2] on as similar a basis as possible.

One caveat about this evaluation should be noted. None of the algorithms were implemented with general support for empty categories, due to the fact that none of the large, independently motivated grammars we had access to contained empty categories. We did, however make use of a grammar transformation (left factoring) that can produce empty categories, but only as the right-most daughter of a rule with at least two daughters. For the algorithms we wanted to test with this form of grammar, we added limited support for empty categories specifically in this position.

3 Terminology and Notation

Nonterminals, which we will sometimes refer to as categories, will be designated by "low order" upper-case letters (A, B, etc.); and terminals will be designated by lower-case letters. We will use the notation a_i to indicate the ith terminal symbol in the input string. We will use "high order" upper-case letters (X, Y, Z) to denote single symbols that could be either terminals or nonterminals, and Greek letters to denote (possibly empty) sequences of terminals and/or nonterminals. For a grammar rule $A \rightarrow B_1 \ldots B_n$ we will refer to A as the mother of the rule and to $B_1 \ldots B_n$ as the daughters of the rule. We will assume that there is a single nonterminal category S that subsumes all sentences allowed by the grammar.

All the algorithms considered here build a collection of data structures representing segments of the input partially or completely analyzed as a phrase of some category in the grammar, which we will refer to as a "chart". We will use the term "item" to mean an instance of a grammar rule with an indication of how many of the daughters have been recognized in the input. Items will be represented as dotted rules, such as $A \rightarrow B_1.B_2$. An "incomplete item" will be an item with at least one daughter to the right of the dot, indicating that at least one more daughter remains to be recognized before the entire rule is matched; and a "complete item" will be an item with no daughters to the right of the dot, indicating that the entire rule has been matched.

We will use the terms "incomplete edge" or "complete edge" to mean an incomplete item or complete item, plus two input positions indicating the segment of the input covered by the daughters that have

[1] Formally, we require that for any m up to the total number parses of the input, we can extract from the chart m parses of a string of length n in time proportional to $m \cdot n$.

[2] We take advantage of Perl 5's ability to arbitrarily nest hash tables and linked lists to produce efficient implementations of the data structures required by the algorithms. In particular, the multi-dimensional arrays required by many of the algorithms are given a sparse-matrix implementation in terms of multiply-nested Perl hash tables.

already been recognized. These will be written as (e.g.) $\langle A \rightarrow B_1 B_2 . B_3, i, j \rangle$, which would mean that the sequence $B_1 B_2$ has been recognized starting at position i and ending at position j, and has been hypothesized as part of a longer sequence ending in B_3, which is classified a phrase of category A. Positions in the input will be numbered starting at 0, so the ith terminal of an input string spans position $i-1$ to i. We will refer to items and edges none of whose daughters have yet been recognized as "initial".

4 Test Grammars

For testing context-free parsing algorithms, we have selected three CFGs that are independently motivated by analyses of natural-language corpora or actual applications of natural language processing. The CT grammar[3] 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 was extracted from the Penn Treebank.[4] We employ a standard test set for each of the three grammars. The test set for the CT grammar is a set of sentences made up by the system designers to test the functionality of the system, and the test set for the ATIS grammar is a randomly selected subset of the DARPA ATIS3 development test set. The test set for the PT grammar is a set of sentences randomly generated from a probabilistic version of the grammar, with the probabilities based on the frequency of the bracketings occuring in the training data, and then filtered for length to make it possible to conduct experiments in a reasonable amount of time, given the high degree of ambiguity of the grammar.

The terminals of the grammars are preterminal lexical categories rather than words. Preterminals were generated automatically, by grouping together all the words that could occur in exactly the same contexts in all grammar rules, to eliminate lexical ambiguity.

	CT Grammar	ATIS Grammar	PT Grammar
Rules	24,456	4,592	15,039
Nonterminals	3,946	192	38
Terminals	1,032	357	47
# Test Sentences	162	98	30
Average Length	8.3	11.4	5.7
# Grammatical	150	70	30
Average # Parses	5.4	940	7.2×10^{27}

Table 1: Grammars and test sets for parser evaluations

Some statistics on the grammars and test sets are contained in Table 1. Note that for the CT and ATIS sets, not all sentences are within the corresponding grammars. The most striking difference among the three grammars is the degree of ambiguity. The CT grammar has relatively low ambiguity, the ATIS grammar may be considered highly ambiguous, and the PT grammar can only be called massively ambiguous.

[3]Courtesy of John Dowding, SRI International
[4]Courtesy of Eugene Charniak, Brown University

5 Left-Corner Parsing Algorithms and Refinements

Left-corner (LC) parsing—more specifically, left-corner parsing with top-down filtering—originated as a method for deterministically parsing a restricted class of CFGs. It is often attributed to Rosenkrantz and Lewis (1970), who may have first used the term "left-corner parsing" in print. Griffiths and Petrick (1965), however, previously described an LC parsing algorithm under the name "selective bottom-to-top"(SBT) parsing, which they assert to be an abstraction of previously described algorithms.

The origins of LC parsing for general CFGs (other than by naive backtracking) are even murkier. Pratt's (1975) algorithm is sometimes considered to be a generalized LC method, but it is perhaps better described as CKY parsing with top-down filtering added. Kay's (1980) method for undirected bottom-up chart parsing is clearly left-corner parsing without top-down filtering, but in adding top-down filtering to obtain directed bottom-up chart parsing, he changed the method significantly. The BUP parser of Matsumoto et al. (1983) appears to be the first clearly described LC parser capable of parsing general CFGs in polynomial time.[5]

LC parsing depends on the left-corner relation for the grammar, where X is recursively defined to be a left corner of A if $X = A$, or the grammar contains a rule of the form $B \rightarrow X\alpha$, where B is a left corner of A. This relation is normally precompiled and indexed so that any pair of symbols can be checked in essentially constant time.

Although LC parsing was originally defined as a stack-based method, implementing it in terms of a chart enables polynomial time complexity to be achieved by the use of dynamic programming; which simply means that if the same chart edge is derived in more than one way, only one copy is retained for further processing. A chart-based LC parsing algorithm can be defined by the following set of rules for populating the chart:

1. For every grammar rule with S as its mother, $S \rightarrow \alpha$, add $\langle S \rightarrow .\alpha, 0, 0\rangle$ to the chart.

2. For every pair of edges of the form $\langle A \rightarrow \alpha.X\beta, i, k\rangle$ and $\langle X \rightarrow \gamma., k, j\rangle$ in the chart, add $\langle A \rightarrow \alpha X.\beta, i, j\rangle$ to the chart.

3. For every edge of the form $\langle A \rightarrow \alpha.a_j\beta, i, j-1\rangle$ in the chart, where a_j is the jth terminal in the input, add $\langle A \rightarrow \alpha a_j.\beta, i, j\rangle$ to the chart.

4. For every pair of edges of the form $\langle A \rightarrow \alpha.C\beta, i, k\rangle$ and $\langle X \rightarrow \gamma., k, j\rangle$ in the chart and every grammar rule with X as its left-most daughter, of the form $B \rightarrow X\delta$, if B is a left corner of C, add $\langle B \rightarrow X.\delta, k, j\rangle$ to the chart.

5. For every edge of the form $\langle A \rightarrow \alpha.C\beta, i, j-1\rangle$, and every grammar rule with a_j as its left-most daughter, of the form $B \rightarrow a_j\delta$, where a_j is the jth terminal in the input, if B is a left corner of C, add $\langle B \rightarrow a_j.\delta, j-1, j\rangle$ to the chart.

An input string is successfully parsed as a sentence if the chart contains an edge of the form $\langle S \rightarrow \alpha., 0, n\rangle$ when the algorithm terminates.

Rules 1–3 are shared with other parsing algorithms, notably E/GHR, but rules 4 and 5 are distinctive to LC parsing. If naively implemented, however, they can lead to unnecessary duplication of work. Rules 4 and 5 state that for every triple consisting of an incomplete edge, a complete edge or input terminal, and a grammar rule, meeting certain conditions, a new edge should be added to the chart.

[5] Cyclic grammars and empty categories were not supported, however.

Inspection reveals, however, that the form of the edge to be added depends on only the complete edge or input terminal and the grammar rule, not the incomplete edge. Thus if this parsing rule is applied separately for each triple, the same new edge may be proposed repeatedly if several incomplete edges combine with a given complete edge or input terminal and grammar rule to form triples satisfying the required conditions. A number of implementations of generalized LC parsing have suffered from this problem, including the BUP parser, the left-corner parser of the SRI Core Language Engine (Moore and Alshawi, 1991), and Schabes's (1991) table-driven predictive shift-reduce parser.

However, if parsing is performed strictly left-to-right, so that every incomplete edge ending at k has already been computed before any left-corner checks are performed for new edges proposed from complete edges or input terminals starting at k, there is a solution that can be seen by rephrasing rules 4 and 5 follows:

4a. For every edge of the form $\langle X \rightarrow \gamma ., k, j\rangle$ in the chart and every grammar rule with X as its left-most daughter, of the form $B \rightarrow X\delta$, if there is an incomplete edge in the chart ending at k, $\langle A \rightarrow \alpha . C\beta, i, k\rangle$, such that B is a left corner of C, add $\langle B \rightarrow X.\delta, k, j\rangle$ to the chart.

5a. For every input terminal a_j and every grammar rule with a_j as its left-most daughter, of the form $B \rightarrow a_j\delta$, if there is an incomplete edge in the chart ending at $j-1$, $\langle A \rightarrow \alpha . C\beta, i, j-1\rangle$, such that B is a left corner of C, add $\langle B \rightarrow a_j.\delta, j-1, j\rangle$ to the chart.

This formulation suggests driving the parser by proposing a new edge from every grammar rule exactly once for each complete edge or input terminal corresponding to the rule's left-most daughter, and then checking whether some previous incomplete edge licenses it via left-corner filtering. If implemented by nested iteration, this still requires as many nested loops as the naive method; but the inner-most loop does much less work, and it can be aborted as soon as one previous incomplete edge has been found to satisfy the left-corner check. Wirén (1987) seems to have been the first to explicitly propose performing left-corner filtering in an LC parser in this way. Nederhof (1993) proposes essentially the same solution, but formulated in terms of a graph-structured stack of the sort generally associated with GLR parsing.

Several additional optimizations can be added to this basic schema. Wirén adds bottom-up filtering (Wirén uses the term "selectivity", following Griffiths and Petrick (1965)) of incomplete edges based on the next terminal in the input. That is, no incomplete edge of the form $\langle A \rightarrow \alpha . X\beta, i, j\rangle$ is added to the chart unless a_{j+1} is a left corner of X. Nederhof proposes that, rather than iterate over all the incomplete edges ending at a given input position each time a left-corner check is performed, compute just once for each input position a set of nonterminal predictions, consisting of the symbols immediately to the right of the dot in the incomplete edges ending at that position, and iterate over that set for each left-corner check at the position.[6] With this optimization, it is no longer necessary to add initial edges to the chart at position 0 for rules of the form $S \rightarrow \alpha$. If P_i denotes the set of predictions for position i, we simply let $P_0 = \{S\}$.

Another optimization from the recent literature is due to Leermakers (1992), who observes that in Earley's algorithm the daughters to the *left* of the dot in an item play no role in the parsing algorithm; thus the representation of items can ignore the daughters to the left of the dot, resulting in fewer distinct edges to be considered. This observation is equally true for LC parsing. Thus, instead of $A \rightarrow B_1B_2.B_3$, we will write simply $A \rightarrow .B_3$. Note that with this optimization, $A \rightarrow .$ becomes

[6] Nederhof proposes several other optimizations, which we evaluated and found not to repay their overhead.

the notation for an item all of whose daughters have been recognized; the only information it contains being just the mother of the rule. We will therefore write complete edges simply as $\langle A, i, j\rangle$, rather than $\langle A \rightarrow ., i, j\rangle$. We can also unify the treatment of terminal symbols in the input with complete edges in the chart by adding a complete edge $\langle a_i, i-1, i\rangle$ to the chart for every input terminal a_i.[7]

Taking all these optimizations together, we can define an optimized LC parsing algorithm by the following set of parsing rules:

1. Let $P_0 = \{S\}$.
2. For every input position $j > 0$, let $P_j = \{B \mid$ there is an incomplete edge in the chart ending at j, of the form $\langle A \rightarrow .B\alpha, i, j\rangle\}$.
3. For every input terminal a_i, add $\langle a_i, i-1, i\rangle$ to the chart.
4. For every pair of edges $\langle A \rightarrow .XY\alpha, i, k\rangle$ and $\langle X, k, j\rangle$ in the chart, if a_{j+1} is a left corner of Y, add $\langle A \rightarrow .Y\alpha, i, j\rangle$ to the chart.
5. For every pair of edges $\langle A \rightarrow .X, i, k\rangle$ and $\langle X, k, j\rangle$ in the chart, add $\langle A, i, j\rangle$ to the chart.
6. For every edge $\langle X, k, j\rangle$ in the chart and every grammar rule with X as its left-most daughter, of the form $A \rightarrow XY\alpha$, if there is a $B \in P_k$ such that A is a left corner of B, and a_{j+1} is a left corner of Y, add $\langle A \rightarrow .Y\alpha, k, j\rangle$ to the chart.
7. For every edge $\langle X, k, j\rangle$ in the chart and every grammar rule with X as its only daughter, of the form $A \rightarrow X$, if there is a $B \in P_k$ such that A is a left corner of B, add $\langle A, k, j\rangle$ to the chart.

Note that in Rule 6, the top-down left-corner check on the mother of the proposed incomplete edge and the bottom-up left-corner check on the symbol immediately to the right of the dot in the proposed incomplete edge are independent of each other, and therefore could be performed in either order. Wirén, the only author we have found who includes both, is vague on the ordering of these checks. For each proposed edge, however, the bottom-up check requires examining an entry in the left-corner table for each of the elements of the prediction list, until a check succeeds or the list is exhausted; while the bottom up check requires examining only a single entry in the left-corner table for the next terminal of the input. It therefore seems likely to be more efficient to do the bottom-up check before the top-down check, since the top-down check need not be performed if the bottom-up check fails. To test this hypothesis, we have done two implementations of the algorithm: LC_1, which performs the top-down check first, and LC_2, which performs the bottom-up check first.

Shann (1991) uses a different method of top-down filtering in an LC parser. Shann expands the list of predictions created by rules 1 and 2 to include all the left-corners of the predictions. He does this by precomputing the proper left corners of all nonterminal categories and adding to the list of predictions all the left-corners of the original members of the list. Then top-down filtering consists of simply checking whether the mother of a proposed incomplete edge is on the corresponding prediction list. Graham, Harrison, and Ruzzo (1980) attribute this type of top-down filtering to Cocke and Schwartz, so we will refer to it as "Cocke-Schwartz filtering". Since our original form of filtering uses the left-corner relation directly, we will call it "left-corner filtering".

[7]Many chart parsers unify the treatment of input terminals and complete edges in this way, by ignoring daughters to the left of the dot, but only for complete edges. The Leermakers optimization permits a unified treatment of incomplete edges, complete edges, and input terminals.

We have implemented Cocke-Schwartz filtering as described by Shann, except that for efficiency in both forming and checking the sets of predictions, we use hash tables rather than lists. The resulting algorithm, which we will call LC_3, can be stated as follows:

1. Let $P_0 = \{\text{all left corners of } S\}$.[8]

2. For every input position $j > 0$, let $P_j = \{\text{all left corners of } B \mid \text{there is an incomplete edge in the chart ending at } j, \text{ of the form } \langle A \rightarrow .B\alpha, i, j\rangle\}$.

3. For every input terminal a_i, add $\langle a_i, i-1, i\rangle$ to the chart.

4. For every pair of edges $\langle A \rightarrow .XY\alpha, i, k\rangle$ and $\langle X, k, j\rangle$ in the chart, if a_{j+1} is a left corner of Y, add $\langle A \rightarrow .Y\alpha, i, j\rangle$ to the chart.

5. For every pair of edges $\langle A \rightarrow .X, i, k\rangle$ and $\langle X, k, j\rangle$ in the chart, add $\langle A, i, j\rangle$ to the chart.

6. For every edge $\langle X, k, j\rangle$ in the chart and every grammar rule with X as its left-most daughter, of the form $A \rightarrow XY\alpha$, if $A \in P_k$, and a_{j+1} is a left corner of Y, add $\langle A \rightarrow .Y\alpha, k, j\rangle$ to the chart.

7. For every edge $\langle X, k, j\rangle$ in the chart and every grammar rule with X as its only daughter, of the form $A \rightarrow X$, if $A \in P_k$, add $\langle A, k, j\rangle$ to the chart.

There is one simple refinement, not mentioned by Shann, that we can add to this algorithm. Since we already have the information needed to perform bottom-up filtering, we can apply bottom-up filtering to building the prediction sets, omitting any left-corner of an existing prediction that is incompatible with the next terminal of the input. This will certainly save space, and may save time as well, depending on the relative costs of adding a nonterminal to the prediction set compared to performing the bottom-up left-corner check. Our modification of LC parsing with Cocke-Schwartz filtering to include this refinement is implemented as LC_4.

	CT Grammar	ATIS Grammar	PT Grammar
LC_1	4.3	15.6	45.0
LC_2	3.4	11.9	43.0
LC_3	3.1	11.6	41.8
LC_4	2.7	11.8	42.3

Table 2: LC parsing algorithm performance comparisons

The results of running algorithms LC_1–LC_4 appear in Table 2. The numbers are CPU time in seconds required by the parser to completely process the standard test set associated with each grammar.[9] LC_2, which performs the bottom-up left-corner check on proposed incomplete edges before top-down left-corner check, is faster on all three grammars than LC_1, which performs the checks in the reverse order—substantially so on the CT and ATIS grammars. Comparing LC_3 with LC_4—which both use Cocke-Schwartz filtering, but differ as to whether the prediction sets are bottom-up filtered—the results are less clear. LC_4, which does filter the predictions, is noticably faster on the CT grammar, while LC_3 which does not filter predictions is slightly faster, but not significanly so, on the ATIS grammar and PT grammar. Finally, both parsers that use Cocke-Schwartz filtering are faster on all grammars than either of the parsers that use left-corner filtering.

[8] Recall that by our definition, the left-corner relation is reflexive so S will be included.

[9] All timings in this report are for execution on a Dell 610 workstation with Pentium III Xeon 550 MHz processors running Windows 2000.

6 Grammar Transformations

One other issue remains to be addressed in our examination of LC parsing. It is a common observation about left-to-right parsing, that if two grammar rules share a common left prefix, e.g., $A \rightarrow BC$ and $A \rightarrow BD$, many parsing algorithms will duplicate work for the two rules until reaching the point where they differ. A simple solution often proposed to address the problem is to "left factor" the grammar. Left factoring applies the following grammar transformation repeatedly, until it is no longer applicable:

> For each nonterminal A, let α be the longest nonempty sequence such that there is more than one grammar rule of the form $A \rightarrow \alpha\beta$. Replace the set of rules $A \rightarrow \alpha\beta_1, \ldots, A \rightarrow \alpha\beta_n$ with $A \rightarrow \alpha A', A' \rightarrow \beta_1, \ldots, A' \rightarrow \beta_n$, where A' is a new nonterminal symbol.

Left factoring has been explored in the context of generalized LC parsing by Nederhof (1994), who refers to LC parsing with left factoring as PLR parsing. Shann (1991) also applies left factoring directly in the representation of the rules he uses in his LC parser, e.g. $A \rightarrow B(C, D)$.

One complication associated with left factoring is that if the daughters of one rule are a proper prefix of the daughters of another rule, then empty rules will be introduced into the grammar, even if there were none originally. For example $A \rightarrow BC$ and $A \rightarrow BCD$ will be replaced by $A \rightarrow BCA', A' \rightarrow D, A' \rightarrow \epsilon$. To explore the cost of this additional complication we compare full left factoring with the following restricted form of left factoring;

> For each nonterminal A, let α be the longest nonempty sequence such that there is more than one grammar rule of the form $A \rightarrow \alpha\beta$, for some nonempty string β. Replace the set of rules $A \rightarrow \alpha\beta_1, \ldots, A \rightarrow \alpha\beta_n$ with $A \rightarrow \alpha A', A' \rightarrow \beta_1, \ldots, A' \rightarrow \beta_n$, where A' is a new nonterminal symbol.

The requirement that β always be nonempty blocks the introduction of empty productions, so with this transformation $A \rightarrow BC$ and $A \rightarrow BCD$ will be replaced by $A \rightarrow BA', A' \rightarrow CD, A' \rightarrow C$.

Left factoring is not the only transformation that can be used to address the problem of common rule prefixes. Left factoring applies only to sets of rules with a common mother category, but as an essentially bottom-up method, generalized LC parsing does most of its work before the mother of a rule is determined. There is another grammar transformation that seems better suited to LC parsing, introduced by Griffiths and Petrick (1965), but apparently neglected since:

> Let α be a maximal sequence of two or more symbols such that there is more than one grammar rule of the form $A \rightarrow \alpha\beta$. Replace the set of rules $A_1 \rightarrow \alpha\beta_1, \ldots, A_n \rightarrow \alpha\beta_n$ with $A' \rightarrow \alpha, A_1 \rightarrow A'\beta_1, \ldots, A_n \rightarrow A'\beta_n$, where A' is a new nonterminal symbol.

Like left factoring, this transformation is repeated until it is no longer applicable. Griffiths and Petrick do not give this transformation a name, so we will call it "bottom-up prefix merging".

It should be noted that all of these grammar transformations simply add additional levels of nonterminals to the grammar, without otherwise disturbing the structure of the analyses produced by the grammar. Thus, when parsing with a grammar produced by one of these transformations, the original analyses can be recovered simply by ignoring the newly introduced nonterminals, and treating their subconstituents as subconstituents of the next higher original nonterminal of the grammar.

Before we apply our LC parsers to our test grammars transformed in these three ways, we make a few small adjustments to the implementations. First, as noted above, full left factoring requires the ability to handle empty categories, at least as the right-most daughter of a rule. We have created modified versions of LC_1–LC_4 specifically to use with the fully left-factored grammar. Second we note that with a left-factored grammar,[10] the non-unary rules have the property that, given the mother and the left-most daughter, there is only one possibility for the rest of the rule. With a bottom-up prefix-merged grammar, the non-unary rules have the property that, given the two left-most daughters, there is only one possibility for the rest of the rule. We take advantage of these facts to store the indexed forms of the rules more compactly and simplify the logic of the implementations of variants of our parsers specialized to these grammar forms.

		CT Grammar	ATIS Grammar	PT Grammar
LC_1	UTF	4.3	15.6	45.0
LC_1	FLF	7.4	63.5	timed out
LC_1	PLF	6.2	66.2	timed out
LC_1	BUPM	3.6	11.7	34.1
LC_2	UTF	3.4	11.9	43.0
LC_2	FLF	5.1	38.2	timed out
LC_2	PLF	4.2	37.7	timed out
LC_2	BUPM	3.1	7.0	27.0
LC_3	UTF	3.1	11.6	41.8
LC_3	FLF	4.2	12.3	45.4
LC_3	PLF	3.8	12.1	43.6
LC_3	BUPM	5.0	17.1	64.6
LC_4	UTF	2.7	11.8	42.3
LC_4	FLF	3.6	11.9	46.6
LC_4	PLF	3.2	11.7	44.4
LC_4	BUPM	3.2	14.7	63.6

Table 3: LC parsing grammar transformation performance comparisons

The results of applying our four LC parsing algorithms with these three grammar transformations are displayed in Table 3, along with results for the untransformed grammars presented previously. The grammar transformations are desginated by the symbols UTF (untransformed), FLF (fully left-factored), PLF (partially left-factored), and BUPM (bottom-up prefix-merged). We set a time-out of 10 minutes on some experiments, since that was already an order of magnitude longer than any of the other parse times. Several observations stand out from these results. First, in every case but one, partial left factoring out-performed full left factoring. Much more surprising is that, in every case but one, either form of left factoring degraded parsing performance relative to the untransformed grammar. For LC_1 and LC_2, the algorithms that use left-corner filtering, the degradation is dramatic, while for LC_3 and LC_4, which use Cocke-Schwartz filtering, the degradation is very slight in the case of the ATIS and PT grammars, but more pronounced in the case of the CT grammar. On the other hand, bottom-up prefix-merging significantly—in some cases dramatically—speeds up parsing for LC_1 and LC_2, while significantly degrading the performance of LC_3 and LC_4.

Looking at the overall results of these experiments, we see that bottom-up prefix merging reverses the previous advantage of Cocke-Schwartz filtering over left-corner filtering. With bottom-up prefix merging, LC_2 is at least 66% faster on the ATIS grammar and 55% faster on the PT grammar than either LC_3 or LC_4; and it is only 15% slower than LC_4 on the CT grammar, and the same speed as LC_3. Averaging over the three test grammars, LC_2 is 40% faster than LC_3 and 38% faster than LC_4.

[10] either fully left-factored, or partially left-factored using our restricted transformation.

7 Extracting Parses from the Chart

The Leermakers optimization of omitting recognized daughters from items raises the question of how parses are to be extracted from the chart. The daughters to the left of the dot in an item are often used for this purpose in item-based methods, including Earley's original algorithm. Graham, Harrison, and Ruzzo (1980), however, suggest storing with each noninitial edge in the chart a list that includes, for each derivation of the edge, a pair of pointers to the preceding edges that caused it to be derived. This provides sufficient information to extract the parses without additional searching, even without the daughters to the left of the dot.

In fact, we can do even better than this. For each derivation of a noninitial edge, even in the Leermakers representation, it is sufficient to attach to the edge only the mother category and starting position of the complete edge that was used in the last step of the derivation. Every noninitial edge is derived by combining a complete edge with an incomplete edge or production. Suppose $\langle A \rightarrow .\beta, k, j\rangle$ is a derived edge, and we know that the complete edge used in the derivation had category X and start position i. We then know that the complete edge must have been $\langle X, i, j\rangle$, since the complete edge and the derived edge must have the same end position. We further know that the incomplete edge (or production) used in the derivation must have been $\langle A \rightarrow .X\beta, k, i\rangle$, since that is the only item that could have combined with the complete edge to produce the derived edge. In this way, for any complete edge, we can trace back through the chart until we have found all the complete edges for the daughters that derived it. The back-trace terminates when we reach a derived edge that has the same start point as the complete edge it was derived from.

8 Comparison to Other Algorithms

We have compared our LC parsers to efficient implementations of three other important approaches to context-free parsing: Cocke-Kasami-Younger (CKY), Earley/Graham-Harrison-Ruzzo (E/GHR), and generalized LR (GLR) parsing. We include CKY, not because we think it may be the fastest parsing algorithm, but because it provides a baseline of how well one can do with no top-down filtering. Our implementation of E/GHR includes many optimizations not found in the original descriptions of this approach, including the techniques used to optimize our LC parsers, where applicable. In our GLR parser we used the same reduction method as Tomita's (1985) original parser, which results in greater-than-cubic worst-case time complexity, after verifying that a cubic-time version was, in fact, slower in practice, as Tomita has asserted.

	CT Grammar	ATIS Grammar	PT Grammar
LC_2+BUPM	3.1	7.0	27.0
CKY	25.0	7.7	50.9
E/GHR	7.3	8.6	27.7
GLR(0)	3.2	14.0	timed out
LC+follow	2.4	6.6	29.6
GLR(0)+follow	2.3	14.1	timed out
GLALR(1)	3.8	14.7	—

Table 4: Alternative parsing algorithm performance comparisons

Table 4 shows the comparison between these three algorithms, and our best overall LC algorithm.

As the table shows, LC_2+BUPM outperforms all of the other algorithms with all three grammars. While each of the other algorithms approaches our LC parser in at least one of the tests, the LC parser outperforms each of the others by at least a factor of 2 with at least one of the grammars.

The comparison between LC_2+BUPM and GLR is instructive in view of the claims that have been made for GLR. While GLR(0) was essentially equal in performance to LC_2+BUPM on the least ambiguous grammar, it appears to scale very badly with increasing ambiguity. Moreover, the parsing tables required by the GLR parser are far larger than for LC_2+BUPM. For the CT grammar, LC_2+BUPM requires 27,783 rules in the transformed grammar, plus 210,701 entries in the left-corner table. For the (original) CT grammar, GLR requires 1,455,918 entries in the LR(0) parsing tables.

The second part of Table 4 shows comparisons of LC_2+BUPM and two versions of GLR with look ahead. The "LC+follow" line is for LC_2+BUPM plus an additional filter on complete edges using a "follow check" equivalent to the look ahead used by SLR(1) parsing. The "GLR(0)+follow" line adds the same follow check to the GLR(0) parser. This builds exactly the same edges as a GSLR(1) parser would, but allows smaller parsing tables at the expense of more table look ups.[11] With the follow check, the parse times for the CT grammar are substantially reduced, but LC_2+BUPM and GLR remain essentially equivalent, while only small changes are produced for the ATIS and PT grammars.

The final line gives results for GLALR(1) parsing with the CT and ATIS grammars.[12] These results are not directly comparable to the others because the LALR(1) reduce tables for the CT and ATIS grammars contained more than 6.1 million and 1.8 million entries, respectively, and they would not fit in the memory of the test machine along with the other LR tables. Various methods were investigated to obtain timing results by loading only a subset of the reduce tables sufficient to handle the test set. These gave inconsistent results, but in all cases times were longer than for either GLR(0) or GLR(0)+follow, presumably due to additional overhead caused by the large tables, with relatively little additional filtering (5–6% fewer edges). The numbers in the table represent the best results obtained for each grammar.

	CT Grammar	ATIS Grammar
LC_2+BUPM	1.8	11.7
CKY	15.4	13.7
E/GHR	3.2	12.1
GLR(0)	1.8	17.8
LC+follow	1.4	10.9
GLR(0)+follow	1.3	18.1

Table 5: Results with longer sentences

A final set of experiments was performed to address possible concerns that the test sentences in our other experiments were too short, and that our results would not generalize to longer sentences. We selected two modified test sets of CT and ATIS sentences. The CT sentences were the 50 longest sentences covered by our CT grammar in the original CT test set, with an average length of 13.5 words, and an average number of parses of 4.2. The ATIS sentences were the 50 longest sentences covered by our ATIS grammar in the DARPA ATIS3 development test set, with an average length of 20.5 words, and an average number of parses of 4516. The results for the principal methods compared

[11] A GSLR(1) reduce table is just the composition of a GLR(0) reduce table and a follow-check table.
[12] No experiments were done for the PT grammar due to the excessively long time required to compute LALR(1) parsing tables for that grammar, given the expectation that the parser would still time out.

in our original cross-algorithm experiments are given in Table 5. When compared to the results in Table 4, the relative performance of the algorithms remains virtually unchanged.

9 Conclusions

Probably the two most significant results of this investigation are the discoveries that:

- LC chart parsing incorporating both a top-down left-corner check on the mother of a proposed incomplete edge and a bottom-up left-corner check on the symbol immediately to the right of the dot in the proposed incomplete edge is substantially faster if the bottom-up check is performed before the top-down check.

- Bottom-up prefix merging is a particularly good match to LC chart parsing based on left-corner filtering, and in fact substantially outperforms left factoring combined with LC chart parsing in most circumstances.

Moreover we have shown that with these enhancements, LC parsing outperforms several other major approaches to context-free parsing, including some previously claimed to be the best general context-free parsing method. We conclude that our improved form of LC parsing may now be the leading contender for that title.

References

Graham, S.L., M.A. Harrison, and W.L. Ruzzo (1980) "An Improved Context-Free Recognizer," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, pp. 415–462.

Griffiths, T.V., and S.R. Petrick (1965) "On the Relative Efficiencies of Context-Free Grammar Recognizers," *Communications of the ACM*, Vol. 8, No. 5, pp. 289–300.

Kay, M. (1980) "Algorithm Schemata and Data Structures in Syntactic Processing," Report CSL–80–12, Xerox PARC, Palo Alto, California.

Leermakers, R. (1992) "A Recursive Ascent Earley Parser," *Information Processing Letters*, Vol. 41, No. 2, pp. 87–91.

Matsumoto, Y., et al. (1983) "BUP: A Bottom-Up Parser Embedded in Prolog," *New Generation Computing*, Vol. 1, pp. 145–158.

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., and H. Alshawi (1991) "Syntactic and Semantic Processing," in *The Core Language Engine*, H. Alshawi (ed.), The MIT Press, Cambridge, Massachusetts, pp. 129–148.

Nederhof, M.J. (1993) "Generalized Left-Corner Parsing," in *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht, The Netherlands, pp. 305–314.

Nederhof, M.J. (1994) "An Optimal Tabular Parsing Algorithm," in *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, Las Cruces, New Mexico, pp. 117–124.

Pratt, V.R. (1975) "LINGOL — A Progress Report," in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, pp. 422–428.

Rosenkrantz, D.J., and P.M. Lewis (1970) "Deterministic Left Corner Parsing," in *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*, pp. 139–152.

Shann, P. (1991) "Experiments with GLR and Chart Parsing," in *Generalized LR Parsing*, M. Tomita (ed.), Kluwer Academic Publishers, Boston, Massachusetts, pp. 17–34.

Schabes, Y. (1991) "Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars," in *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, California, pp. 106–113.

Tomita, M. (1985) *Efficient Parsing for Natural Language*, Kluwer Academic Publishers, Boston, Massachusetts.

Wirén, M. (1987) "A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing," in *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, Copenhagen, Denmark, pp. 226–233.