METHODS FOR PARALLELIZING SEARCH PATHS IN PARSING

Carl de Marcken
MIT Artificial Intelligence Laboratory
NE43-800b
545 Technology Square
Cambridge, MA, 02139, USA
cgdemarc@ai.mit.edu

20 April 1994

Summary

This paper investigates the application of various efficiency-motivated search techniques to generate-and-test problems of the sort that arise in principle-based parsing. In particular, various means of eliminating redundant computation and performing operations in parallel are examined, and performance improvements are compared with those resulting from the use of more deterministic theories of language processing.

Subject Areas: parallel search, parsing

Word Count: XXX

INTRODUCTION

Many search problems in natural language are solved with simple combinatoric algorithms that unnecessarily duplicate and serialize work at considerable computational expense. There are a number of techniques that can eliminate redundant computations and perform remaining operations in parallel, effectively reducing the branching factors of these algorithms. This paper investigates the application of some of these techniques to the problem of parsing with generate-and-test search, the simplest method for parsing using principles-and-parameters theories, by presenting a language that conveniently represents non-deterministic search problems and several parsers that have been implemented using it. The performance improvements that result from the use of sophisticated search strategies are compared with those resulting from the use of more deterministic theories of language processing. The interested reader can find a more detailed treatment of the ideas in this paper in (de Marcken, 1994).

GENERATE-AND-TEST PARSING

One way of envisioning a grammar is as a set of constraints. In the domain of parsing sentences, grammars constrain parses, sets of relations between words. For certain classes of grammars, such as context-free grammars (but few others), it is known how to efficiently generate only those sets of relations which can satisfy the grammar in question, using Earley's algorithm for instance. But for other classes of grammars used in natural language applications, such as most crafted by Chomskyan linguists, there are no obvious means of efficiently generating only those parses that will be accepted by a given grammar. As a consequence, parsing using such grammars reduces to explicitly treating the grammar as a set of constraints, enumerating parses one by one and checking each against the grammar. Any parse that passes muster is an acceptable interpretation of the sentence in question.

This process of generating and testing parses can be quite inefficient if the search space is large; indeed, merely guaranteeing termination can be difficult. The modular structure of natural language does permit a number of optimizations to the generate-and-test procedure, though. (Fong, 1991) discusses some of these. For instance, one need not enumerate all aspects of a parse before beginning the checking stage: a parser may verify that a parse meets basic phrase-structure constraints before enumerating and testing various pronoun reference possibilities. This prevents the branching factor associated with pronoun reference ambiguity from unnecessarily multiplying out the work done in testing phrase structure constraints. In many traditional systems this optimization is realized through multiple stages of parsing.

Figure 1 presents a dependency diagram of the generators and testers in a version of Fong's parser (Fong, 1991). The topmost generator produces unadorned phrase structure trees. Other generators (some deterministic, some non-deterministic) add pronoun bindings, case, movement chains, and other information to the base trees. The many filters rule out various structures on linguistic grounds. In general, a given filter may depend on any subset of the generators.

In the past, generate-and-test parsers, often implemented using Prolog, have parsed using some linear ordering of generators and filters that satisfies the partial ordering of the module dependency diagram. Fong explores in great depth the tradeoffs between various orderings in his parser. But no matter which ordering is chosen, some unnecessary combinatorial inefficiency results from sequencing two non-deterministic generators that are conceptually independent. For instance, in figure 1, generator 3 (assignment of thematic roles) and generator 6 (LF-movement) are conceptually independent. Ordering one before the other implies that for each non-deterministic branch the first one takes, the second one is going to perform an identical redundant computation. Generators 1 and 4 are also independent of each other, but do have filters that depend on both of them. Therefore at some point the cross product of the values each produces must be computed, but the actual computation

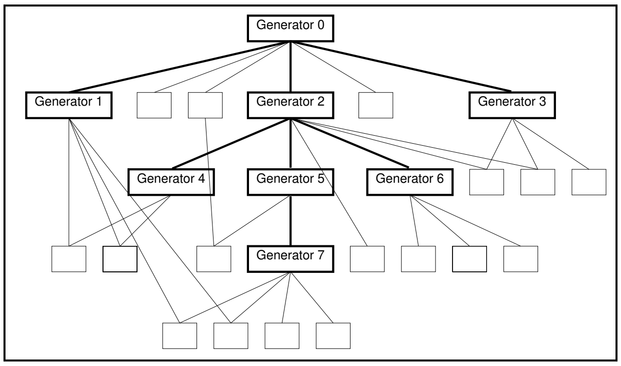

Figure 1: The partial ordering representing the module dependencies in Fong's parser. Dark rectangles represent generators and light ones represent filters.

involved in initially generating these values may be reduced through memoization, and the combinatorics confined to the trivial enumeration of previously-computed results.

THE SEARCH LANGUAGE

A new search language has been implemented that permits generators and filters in search problems that have no dependencies to be computed independently. Based on SCREAMER (Siskind et al., 1993), a non-deterministic extension to COMMON LISP, it encodes search problems as conveniently as PROLOG does, and has special features which permit memoization of side-effecting functions, concurrent execution of independent modules, user-declared data dependencies, and other facilities for improving the efficiency of generate-and-test search.

SCREAMER

SCREAMER is a non-deterministic extension to COMMON LISP. It uses CPS-conversion to implement an efficient **either** function, which non-deterministically returns one of its arguments. Thus

```
(let ((x (either 0 2))
      (y (either 0 1)))
  (if (>= x y) (list x y) (fail)))
```

will non-deterministically return **(0 0)**, **(2 0)**, and **(2 1)**. SCREAMER undoes side-effects upon backtracking and provides means for consolidating values.

```
(defgenerator ANAPHOR?
     ((tree PHRASE-STRUCTURE))
  (map-over-tree-phrases (phrase tree)
    (when (empty-noun-phrase? phrase)
      (set! (is-anaphor? phrase)
            (either 'T 'nil)))))

(deffilter EMPTY-CASE-FILTER
     ((tree PHRASE-STRUCTURE)
      ANAPHOR? CASE-ASSIGNMENT)
  (map-over-tree-phrases (phrase tree)
    (when (empty-noun-phrase? phrase)
      (if (or (and (is-anaphor? phrase)
                   (has-case? phrase))
              (and (not (is-anaphor? phrase))
                   (not (has-case? phrase))))
          (reject)))))
```

Figure 2: The **anaphor?** generator is dependent on the tree produced by the **phrase-structure** generator. For every empty noun-phrase it non-deterministically specifies whether the phrase is or is not anaphoric. The **empty-case-filter** uses the results of the **phrase-structure**, **anaphor?** and **case-assignment** generators to reject parses in which a phrase violates the condition that anaphoric empty categories must not receive case while non-anaphoric ones must.

Module Specification

Generators and filter modules are specified in the language using **defgenerator** and **deffilter** macros. The input arguments to a module are the names of other generators, the parents of the module in the module dependency graph. Figure 2 presents simplified versions of two modules.

Search Strategy Specification

It is not enough to specify a full set of generators and testers, and their dependencies. Before a search can begin, a strategy must be specified. As an example, a traditional depth-first search strategy is implemented by

```
(defsearcher DFS-PARSE
  (generate PHRASE-STRUCTURE
    (filter ...)
    (generate ANAPHOR?
```

```
      (filter ...)
      (generate CASE-ASSIGNMENT
        (filter ...)
        (filter EMPTY-CASE-FILTER)))))
```

This search strategy is inefficient in that the **case-assignment** generator is executed once for every value the **anaphor?** generator produces. The independence of these two generators can be reflected in the search strategy by changing it to

```
(defsearcher MODULAR-PARSE
  (generate PHRASE-STRUCTURE
    (filter ...)
    (generate ANAPHOR?
      (filter ...))
    (generate CASE-ASSIGNMENT
      (filter ...))
    (filter EMPTY-CASE-FILTER)))
```

Using this revised strategy, the anaphoric and case properties of phrases are determined independently, and the various possible values stored. Only when the doubly-dependent **empty-case-filter** is applied is the cross-product of these two value sets computed. This process is described in more detail below.

When a generator is executed it encapsulates each output value with closures representing the side-effects performed during the computation of that value. This enables a search program to efficiently reproduce the entire effect of a generator. Each value output by a generator is also linked to the generator's input argument (value), to enable various forms of failure propagation. To enable failure propagation even in the case where a module has several input arguments, as **empty-case-filter** does, a single "virtual" input value is created by combining the values from each parent in the module dependency graph. As a generator produces values, it stores them in a *set*. Thus in the **modular-parse** example above, when it comes time to execute **empty-case-filter** the sets produced by **case-assignment** and **anaphor?** can be efficiently combined and enumerated (without actually re-executing the generators) and input to the filter. When a filter rejects its input argument, that value is permanently deleted from the set it is contained in. Should any generator's set of output values become empty, the input argument that produced that set is recursively deleted.

Memoization

Natural language problems are characterized by large numbers of local ambiguities, often represented by many large data structures built from common subcomponents. A given subcomponent may have the same redundant operations performed on it for each structure it appears in, an inefficiency eliminated in part through *memoization*, the storing of the results of an operation for future reuse. A form of memoization is used by every generator to permit rapid re-enumeration. The language has facilities for efficiently memoizing any function.

Memoization is not always as productive as might be thought. It must be possible to efficiently encode the environment in which a function is executed before that function can be memoized. For instance, one could not memoize the **empty-case-filter** on the phrase argument without taking into account the anaphor and case bindings present. But memoization does have some pleasant side effects: when a memoized function is called with the same arguments it repeatedly returns the same value. Thus further computations memoized on the result of the function will be more efficient, and subcomponent failures (see the next section) have wider-ranging effects.

Component Failures

It may be that a filter rejects a parse because some particular subcomponent of that parse violates a linguistic condition. That component may be shared among many parses, as is common in ambiguous sentences. The language has a facility for specifying failure dependency chains (for instance, when a phrase is rejected then any phrase containing that phrase as a daughter is similarly rejected). A filter may reject a particular component of a parse, which has the effect of blocking that parse and all other parses that include the same component.

Concurrency

In the search language, non-interacting modules are made computationally independent. It is a natural extension to make them computationally concurrent. Although the language has not been implemented on a concurrent computer system, the compiler can produce code with the necessary structure for concurrent execution, to the point that the code can be run in a simulated parallel environment such as LUCID COMMON LISP's processes. Concurrent search execution is specified through **cobegin**, **pgenerate** and **pfilter** commands. For instance,

```
(defsearcher PARALLEL-PARSE
  (pgenerate PHRASE-STRUCTURE
    (cobegin
      (filter ...)
      (pgenerate ANAPHOR?
        (filter ...))
      (pgenerate CASE-ASSIGNMENT
        (filter ...)))
    (pfilter EMPTY-CASE-FILTER)))
```

specifies that each phrase structure is to be examined by a separate, conccurant process, each one generating the various case and anaphor assignments in parallel. For a given phrase structure, once all the case and anaphor assignments have been computed, the **empty-case-filter** is applied to all combinations concurrently.

The most significant change necessary to accommodate such parallelism is the elimination of conflicting side effects from the language, easily accomplished by transforming lookups into searches through the closure lists created during the memoization of side-effecting procedures. As yet the language has only be run in a simulated concurrent environment where no explicit inter-process communication needed to take place, so many fundamental complexities have been temporarily sidestepped.

THE PARSERS

The goal of this research was not to produce a particularly efficient, complete, or psychologically plausible implementation of any theory of language, but to test the viability of various efficiency-motivated search techniques. To this end, several complex (though linguistically incomplete) parsers were implemented in the search language and tested under various search strategies. The linguistic theory reflects (Chomsky, 1981).

Each parser implements a basic X-Bar theory of phrase structure, and further annotates trees with case markers, theta roles, D-structure to S-structure movement chains, and binding indices. Figure 3 presents the modules of one of the test parsers. The variation between the three parsers stems from different theories of movement chains and the specification of the anaphoric and pronominal aspect of empty categories. One parser generates the anaphoric and pronominal specification of empty categories using the non-deterministic algorithm of figure 2; the other two use the deterministic algorithm presented in (Correa, 1991). One of these two also uses the deterministic chain formation algorithm from (Correa, 1991)- the other two use a non-deterministic algorithm. The parsers that use Correa's algorithms have slightly different data dependencies from the ones shown in figures 2 and 3. Correa's algorithms are good approximations to the fully non-deterministic algorithms but do have various descriptive inadequacies.

In each parser it is the non-deterministic generators, those branching generators that necessitate search, that can be blamed for most of the computation involved in a parse. The deterministic generators and the filters are essentially a constant factor in every parse. The sources of non-determinism are: the initial phrase structure generator (which also inserts empty categories); the calculation of movement chains; the specification of the anaphoric and pronominal nature of each empty category; and the generation of binding-theory indices.

RESULTS

As we are concerned with an abstract notion of computation, and not the specifics of memory and cycle expenditure, each parser was evaluated by recording the number of values fed to (and output by) each module over a small set (15) of test sentences. For instance, figure 4 presents the input-output data for the parser that does

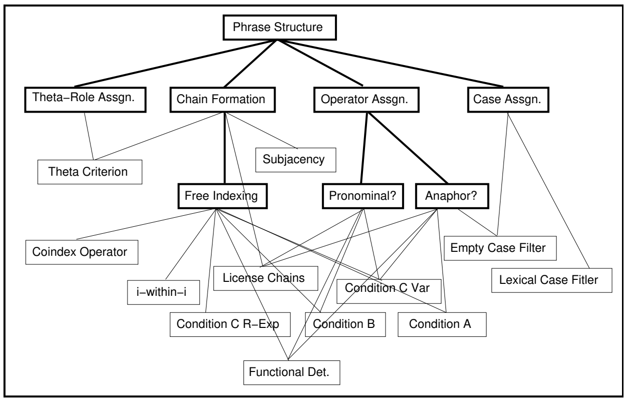

Figure 3: The partial ordering representing the module dependencies in the test parsers. Dark rectangles represent generators and light ones represent filters. non-deterministicgenerators appear in bold face.

not use Correa's deterministic algorithms, under both a modularized and a depth-first search strategy.

Comparing the performance of the modularized search with the depth-first search, there is a notable advantage to modularization (3775 vs. 5323). This is because the depth-first-search parser executes the **pronominal?** module (and others) many more times than is necessary. In the modular version, **pronominal?** is only executed once on each value output by **operator-assignment** (that has not already been deleted), 23 times compared to the 216 of DFS, which folds in all other generators' effects, thus multiplying the number of values. No matter how the depth-first search is rearranged, some generators and filters will be executed more times than is necessary. The fundamental reason that modularization of the search process does not further improve performance is that, in the implemented linguistic theory, the generators with the highest branching factors are all in a single dependency chain. Changes in the parser quickly improve the relative efficiency of DFS search. For the parser that uses Correa's deterministic empty-category determination algorithm, the numbers are 3095 vs. 2732, and for the parser that uses both Cor-

Module	Mod.		DFS	
	in	out	in	out
Phrase Structure	15	80	15	80
Θ-Role Assgn.	80	43	80	43
Operator Assgn.	43	43	43	43
Case Assgn.	43	43	43	42
Lexical Case Filter	42	40	42	40
Anaphor?	40	85	40	54
Empty Case Filter	54	40	54	40
Chain Formation	40	221	40	221
Subjacency	221	217	221	217
Theta Criterion	217	35	217	35
Free Indexing	35	297	35	297
Condition C R-Exp	297	242	297	242
Coindex Operator	242	234	242	234
i-within-i	234	234	234	234
Condition A	234	216	234	216
Pronominal?	23	56	216	1022
Condition B	1022	727	1022	727
License Chains	99	19	727	131
Condition C Var	146	69	131	62
Functional Det.	62	27	62	27
Totals	3775		5323	

Figure 4: The computation performed by each module during the parse of 15 sentences by a modular and a depth-first search strategy. If we assume the resources expended by a module are equal to the maximum of the number of values input to and the number of values output by the module, then the total resource expenditure over the 15 sentences is 3775 and 5323.

rea's empty-category and chain-formation algorithms, there are no independent non-deterministic generators and therefore there is no inefficiency associated with DFS search. Depth-first search actually becomes substantially more efficient that a modular approach if only one parse is required instead of a full search. The modular parsers have essentially computed every value before any is output, whereas DFS can stop before many avenues have been explored. Thus, asking for only one parse per sentence does not improve the performance of a modular search but DFS improves from 5323 to 3526.

The component failure propagation mechanism the search language provides does indeed improve performance. Although the test sentences are not particularly ambiguous, the case filters (for both lexical categories and empty categories) can localize failures and reject all parses that either do not assign case to a specific lexically realized noun phrases, or empty categories that have been marked anaphoric and have been assigned case. In figure 4 that, for instance, the **anaphor?** generator outputs 85 values but the **empty-case-filter** only sees 54 of these, because as it rejects some parse many others are simultaneously deleted.

Memoization of various functions improves performance in two ways: both by reducing the resource expenditure of a function that is called multiple times with the same arguments, and also by improving the performance of the component failure feature of the search language. Of the 7 functions memoized (assignment of case, theta role, anaphoric and pronominal indices; computation of c-command; computation of binding indices; and chain formation) there was a 74% reduction in the number of procedure calls (33701 to 8870), 62% of the remainder being recalls and 38% original computations. There was also a small but noticeable improvement in our coarse measure of resource utilization, from 3798 without memoization to 3775, because the component failure system took advantage of the shared values produced by the memoized functions. A much greater improvement would have been seen had the phrase structure generator not been a tabular parser, inherently memoized.

Concurrent execution of different modules obviously holds the greatest potential for performance improvement. If one measures the cost of a module as the maximum number of values input to it or output by it, and presumes that the cost of process creation is zero, then by our simplistic measure the cost of computation reduces from 3775 to 594 immediately for the modular search strategy, and from 5323 to 588 under the DFS strategy for the most non-deterministic parser. The parser that utilizes both Correa's deterministic algorithms improves from 823 to 410 (modular) and 824 to 410 (DFS). If different independent modules are run concurrently with **cobegin** (in addition to running a given module on several values at once), parse times are governed by the cost of a single search path through the longest chain of dependent modules, an almost negligible sum. Of course, these numbers are misleading, since they do not reflect the profound effects different computing architectures and consequent processing overheads would have on a concurrent implementation. For now, these number illustrate two points: with unlimited concurrency the non-deterministic parsers are little less efficient than the deterministic ones; and concurrency holds extraordinary potential for improving the efficiency of generate-and-test parsing.

CONCLUSIONS

There is a lot of moderately interesting information to be gleaned from tests similar to the ones described here. Rather than exploring such details, we present two broad generalizations from this work. First, the language makes it very easy to implement generate-and-test search problems, especially this particular brand of principles-and-parameters parsers. Second, modularity in search, error propagation mechanims, memoization and other special techniques improve the efficiency of a parser implementation to some extent or other, but their small to medium benefits pale in respect to the effect of a change to the problem itself, such as the substitution of deterministic linguistic modules for modules with high branching factors. The exception to this second remark is concurrent execution. This research demonstrates that there is no reason in theory that generate-and-test parsers can not be made to run extremely efficiently on massively parallel processors, but leaves the details of such an implementation incomplete.

REFERENCES

Noam Chomsky. 1981. *Lectures on Government and Binding.* Foris.

Nelson Correa. 1991. Empty categories, chain binding, and parsing. In *Principle-Based Parsing: Computation and Psycholinguistics*, Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors. Kluwer Academic Publishers.

Carl de Marcken. 1994. Methods for Parallelizing Search Paths in Parsing. Technical Report 1453, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts.

Sandiway Fong. 1991. *Computational Properties of Principle-Based Grammatical Theories.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Jeffrey Siskind and David McAllester. 1993. SCREAMER: a portable efficient implementation of nondeterministic COMMON LISP. Technical Report IRCS-93-03, University of Pennsylvania Institute for Research in Cognitive Science.