COLING 82, J. Horecký (ed.)
North-Holland Publishing Company

AN IMPROVED LEFT-CORNER PARSING ALGORITHM

Kenneth M. Ross

Computer Science Laboratory
Central Research Laboratories
Texas Instruments Incorporated
Dallas, Texas
U.S.A.

This paper proposes a series of modifications to the left corner parsing algorithm for context-free grammars. It is argued that the resulting algorithm is both efficient and flexible and is, therefore, a good choice for the parser used in a natural language interface.

INTRODUCTION

Griffiths and Petrick (1965) propose several algorithms for recognizing sentences of context-free grammars in the general case. One of these algorithms, the NBT (Non-selective Bottom to Top) Algorithm, has since been called a "left-corner" algorithm. Of late, interest has been rekindled in left-corner parsers. Chester (1980) proposes a modification to the Griffiths and Petrick algorithm which "combines phrases before it has found all of their components." Slocum (1981) shows that a left-corner parser inspired by Griffiths and Petrick's algorithm and by Chester's performs quite well when compared with parsers based on a Cocke-Kasami-Younger algorithm (see Younger 1967).

This paper will propose modifications to Griffiths and Petrick's NBT algorithm which result in a more efficient parsing algorithm. A selective version of this new algorithm has been implemented in Maclisp on a DEC 2060 and in Lisp Machine Lisp on an LMI Lisp Machine. It is being used as the context-free component of a parser being used to build natural language interfaces at Texas Instruments. This algorithm is like the NBT algorithm and differs from Chester's in that it does not require the grammar to be in a special format. Any rule of a context-free grammar is acceptable. The new algorithm builds on the Griffiths and Petrick algorithm and is an extension of the algorithm proposed in Ross (1981).

The algorithm given in Griffiths and Petrick (1965) (henceforth G+P) is a recognition algorithm, not a parsing algorithm. Thus, it will only indicate whether or not a string can be produced from a grammar. It will not produce a parse tree. Although algorithms to recognize or parse context-free grammars can be stated in terms of push-down store automata, G+P state their algorithm in terms of turing machines because the algorithm is easier to understand in these terms. A somewhat modified version of their algorithm will be given in the next section. These modifications transform the algorithm into a parsing algorithm and also simplify it a bit.

The G+P algorithm employs two push down stacks. The modified algorithm to be given below will use three, called alpha, beta and gamma. Turing machine instructions are of the following form, where A,B,C,D,E and F can be arbitrary strings of symbols from the terminal and nonterminal alphabet.

```
[A,B,C] --> [D,E,F] if "Conditions"
```

This is to be interpreted as follows:

If A is on top of stack alpha,
B is on top of stack beta,
C is on top of stack gamma,
and "Conditions" are satisfied
then replace A by D, B by E, and C by F.

THE NBT ALGORITHM

The NBT algorithm is a nonselective version of the SBT (Selective Bottom to Top) algorithm, also given in G+P. The only difference between the two is that the SBT algorithm employs a reachability matrix to selectively eliminate bad paths before trying them. For more on this, see G+P and Ross (1981). For the purpose of this paper, it is not necessary to say anything more than that the addition of a reachability matrix modifies the algorithm only slightly and serves only to make the algorithm more efficient.

A version of NBT modified to employ a third stack and to parse rather than recognize strings follows. This algorithm will be modified further throughout the paper.

(1) [V1,X,Y] --> [Ø,V2 ... Vn,t X,A Y]
if A --> V1 V2 ... Vn is a rule of
the phrase structure grammar
X is in the set of nonterminals and
Y is anything

(2) [X,t,A] --> [A X,Ø,Ø]
if A is in the set of nonterminals

(3) [B,B,Y] --> [Ø,Ø,Y]
if B is in the set of nonterminals or
terminals

To begin, put the terminal string to be parsed followed by END on stack alpha. Put the nonterminal which is to be the root node of the tree to be constructed followed by END on stack beta. Put END on stack gamma. The symbol t is neither a terminal nor a nonterminal. If END is on top of each stack, the string has been recognized. If none of the turing machine instructions apply and END is not on the top of each stack, the path which led to this situation was a bad path and does not yield a valid parse.

The rules necessary to give a parse tree can be stated informally (i.e., not in terms of turing machine instructions) as follows:

When (1) is applied, attach V1 beneath A.

When (3) is applied, attach the B on alpha B as the
right daughter of the top symbol on gamma.

Note that there is a formal statement of the parsing version of NBT in Griffiths (1965). However, it is somewhat more complicated and obscures what is going on during the parse. Therefore, the informal procedure given above will be used instead.

Intuitively, what NBT does is put the symbols on alpha together in a bottom-up manner with the ultimate goal of constructing a tree that has, at its top, whatever nonterminal symbol is on top of beta. So, to parse a sentence of English, NBT would begin with the lexical categories of the words to be parsed as a sentence on alpha and the nonterminal "S" on beta. An application of turing machine instruction (1) reduces this problem to a simpler one. (1) finds some phrase structure rule containing the symbol that is on top of alpha immediately

after the arrow. So, if the first symbol on alpha was "det", the phrase structure rule NP --> det AdjP N would qualify. By this application of (1), the problem is reduced to building an AdjP and finding an N from the symbols on alpha. Once this is done, the trees for the "det", the "AdjP" and the "N" would be combined into an NP. By application of (2), the NP would be put on alpha. Then a rule with NP immediately following the arrow would be looked for so that (1) could apply again.

NBT is a nondeterministic algorithm. The nondeterminism comes from two places. Firstly, rule (1) can apply in more than one way. For this to happen, there would need to be two phrase structure rules with the same nonterminal symbol immediately after the arrow. The following two rules are an example of this.

```
X --> Y Z1 Z2 Z3
R --> Y X1 X2
```

Secondly, rule (3) and rule (1) could apply in the same situation. Intuitively, an application of rule (3) indicates that a tree topped by node B was being searched for and a tree topped by node B has been found, so use the tree just found as the tree that was sought. Rule (1) could apply as well if a phrase structure rule of the form X --> B Y1 Y2 ... Yn existed. Applying (1) indicates that the B being sought is not the B that was just built. Rather, the B that was just built is an initial subtree of the B being sought.

RULES WITH ABBREVIATIONS

An important aspect of the modified algorithm being proposed is that it can deal directly with rules which employ abbreviatory conventions which are utilized by linguists. Thus, parentheses (expressing optional nodes) and curly brackets (expressing the fact that one of the set of nodes in brackets should be chosen) can appear in the rules that the parser accesses when parsing a string.

Assume that left and right parentheses are put on stack beta as separate elements. Also assume that left and right curly brackets are put on stack beta as separate items. Given these assumptions, to modify NBT to handle rules with parenthesized elements, the following turing machine instructions must be added.

```
(4) [X, ( C1 C2 ... Cn ),Y] --> [X,C1 C2 ... Cn,Y]
(5) [X, ( C1 C2 ... Cn ),Y] --> [X,Ø,Y]

    For all i, Ci = ( Cj Cj+1 ... Cp ) or
                    { C1 C1+1 ... Cm } or
                    X
                    if X is in the set of terminals.
```

The first rule will apply when the parenthesized node is present. The second rule will apply when the node is not present. The Ci variable handle cases of nested parentheses or curly brackets. Informally, a Ci is a variable that stands for a nonterminal, a terminal, a left parenthesis followed by some number of expressions which are Ci's followed by a right parenthesis, or a left curly bracket followed by some number of expressions which are Ci's followed by a right curly bracket.

The following rules are necessary to directly parse with rules containing curly brackets.

```
(6) [X,{ C1 X,Y] --> [X,{,Y]
(7) [X,{ C1 X,Y] --> [X,C1 X,Y]
                     if X not = }
(8) [X,{ },Y] --> [X,Ø,Y]
(9) [X, { C1 },Y] --> [X,C1,Y]
```

(10) [X,: C1,Y] --> [X,:,Y]

where : is a special symbol which is
neither a terminal or a nonterminal symbol,

C1 is a Ci type variable as defined earlier.

Once these modifications are incorporated, the resulting algorithm will be more efficient than if the NBT algorithm were used with abbreviated rules completely expanded into many distinct rules. To see why this is so, consider a situation in which there was a rule of the form X --> A1 A2 ... An (Z). If this was replaced by two rules, X --> A1 A2 ... An Z and X --> A1 A2 ... An, the parse would have to be split immediately upon encountering X. However, if the alternative solution being proposed were used, rather than parsing for A1, A2, ..., to An twice, they would only be parsed for once. The parse path would not split until it came time to decide whether we wanted to look for Z or not. In general, every rule which has, following the arrow, some number of obligatory elements followed by a parenthesized element will result in a savings. Thus, any such rule can be parsed with more efficiently than the two rules it would be turned into if parentheses were eliminated. Note that the additional cost here is quite small. For each parenthesized element, (4) and (5) will each apply once. In the alternative solution, many rules might apply unnecessarily to the parse for the nodes which came before the parenthesized node.

There is a class of grammars for which the solution proposed here will require a bit more work than the solution where parentheses are simply eliminated from the grammar. These are grammars that only have rules in which parenthesized items come first and have no rules in which obligatory items precede optional ones. In a grammar with both kinds of rules, the savings made far outweigh the amount of extra work needed. Since the classes of grammars used in parsing systems generally have both kinds of rules, my solution will result in a savings for these. Note that a similar efficiency argument can be made for the curly bracket case.

The above rules will handle all occurrences of parentheses and curly brackets except for those in which the item immediately following the arrow in a phrase structure rule is in parentheses or curly brackets. The algorithm could be modified to handle these cases directly, however, this will not increase efficiency. Items in curly brackets or parentheses that immediately follow the arrow in a phrase structure rule must be expanded immediately upon encountering them. There is no savings in postponing this expansion until run time. Putting off the choice of how to expand such a phrase structure rule will not allow paths to be merged together. Therefore, the best way to handle these is to expand all such rules into rules that do not have this property.

Linguistically, the above is an interesting result. Linguists have claimed that use of parentheses simplifies the grammar. Since simpler grammars are preferred to more complex ones, a solution which collapses two rules to one by parentheses is preferable to a solution that has two distinct rules. In parsing, we see that in many instances, the use of one rule with parentheses rather than two rules without results in the parser operating more efficiently. It is able to merge parse paths together which would have been distinct had several context-free rules not been collapsed together as one, using the abbreviatory conventions. Thus, a notational device which was originally proposed to simplify phrase structure rules actually results in a more efficient parse in many cases. Therefore, at least for some cases, we have additional evidence for the use of parentheses in phrase structure rules.

DEPTH OR BREADTH FIRST?

There has of yet been no discussion of the order in which the algorithm proceeds. The statement of the algorithm is completely neutral in this respect. However, an implementation must impose some control structure. When a parse is started, there is one 3-tuple containing the information on stacks alpha, beta, and gamma. In general, there are many different rules of the parsing algorithm that can be applied after this point. In order to assure that all possible paths are pursued to completion, it is necessary to proceed in a principled way.

One strategy is to push one state as far as it will go. That is, apply one of the rules that are applicable, get a new state, and then apply one of the applicable rules to that new state. This can continue until either no rules apply or a parse is found. If no rules apply, it was a bad parse path. If a parse is found, it is one of possibly many parses for the sentence. In either case, the algorithm must continue on and pursue all other alternative paths. An easy way to do this and assure that all alternatives are pursued is to backtrack to the last choice point, pick another applicable rule, and continue in the manner described earlier. By doing this until the parser has backed up through all possible choice points, all parses of the sentence will be found. A parser that works in this manner is a depth-first backtracking parser. This is probably the easiest control structure to use for a left-corner parser.

Alternative control structures are possible. For instance, rather than pursuing one path as far as possible, one could go down a parse path to some desired distance, save that state for later, and come back up to the top and start some other parse path. The original parse path could be pursued later from the point at which it was stopped. The problem with such an approach is keeping track of all the options.

In the algorithm being proposed here, the decision of whether the parse proceeds in a depth-first or breadth-first manner is governed by a parameter which is adjustable. Thus, the parser can proceed to a setable depth down each parse path before going off and pursuing others. This mechanism works by saving the state of the parser when it reaches the desired depth down a particular parse path. Once all paths are pursued to this depth, the parser is called again with each of the states that were saved.

To enable the parser to function as described above, the control structure for a depth-first parser described earlier is used. To introduce the ability to proceed in a breadth-first manner, the parser is only given a subset of the input string. Then, the item MORE is inserted after the last item that is given to the parser. If no other instructions apply and MORE is on top of stack beta, the parser must begin to backtrack as described earlier. Additionally, the state must be saved. Once all backtracking is completed, more input is put on beta and parsing begins again with each of the saved states.

By changing the amount of input that is given, the degree to which the parser proceeds either depth or breadth first can be controlled. If one word is given at a time, the parser is completely breadth-first. If the entire sentence is given, it is completely depth-first. Any other amount results in some combination of the two.

This mechanism enables the algorithm to easily incorporate a well-formed substring table. All that needs to be done is compare the set of saved states and merge the ones that have subgoals in common. By setting the parameter to different values, the degree to which the well-formed substring table is used can be controlled. This is particularly important in light of Slocum's results which indicate that the overhead involved in maintaining such a table can exceed the savings that it

gives. By having the degree to which the table is used be adjustable, the proper setting can be determined, based on the grammar and the sorts of queries that are asked most often.

Additionally, the algorithm can be used to process the sentence word by word as it is typed in. When used as the parser in a natural language interface, this can increase the speed of a parse since work can proceed as the user is typing and composing his input.

Bibliography

[1] Chester, D., A parsing algorithm that extends phrases, American Journal of Computational Linguistics, 6-2 (1980) 87-96.

[2] Griffiths, T., On procedures for constructing structural descriptions for three parsing algorithms, Communications of the ACM, 8 (1965) 594.

[3] Griffiths, T. and Petrick, S.R., On the relative efficiencies of context-free grammar recognizers, Communications of the ACM, 8 (1965) 289-300.

[4] Ross, K., Parsing English phrase structure, Ph.D. Dissertation, Dept. of Linguistics, Univ. of Mass. (Sept. 1981).

[5] Slocum, J., A practical comparison of parsing strategies, Proceedings of the 19th Annual Meeting of the ACL, (1981) 1-6.

[6] Younger, D., Recognition and parsing of context-free language in time n3, Information and Control 10 (1967) 189-208.