Minimum Distance Error Correction

Julia Dain

Dept of Computer Science
University of Warwick
Coventry CV4 7AL
UK

ABSTRACT

A method is presented for incorporating error correction using a minimum distance measure into LR parsers. The method is suitable for use by an automatic parser-generator. State information in the LR parser stack at the point of detection of error is used to generate a set of strings which are potential repairs to the input. A string with least minimum distance from the actual input is chosen and the parser is restarted. Practical methods for generating the set of repairs are discussed.

June 1987

Minimum Distance Error Correction

Julia Dain

Dept of Computer Science
University of Warwick
Coventry CV4 7AL
UK

Introduction

We investigate a method for incorporating error correction into practical parsers for context-free languages. We are specifically interested in the LR parsers for a number of reasons. Firstly, the LR grammars generate a wide class of languages, the deterministic context-free languages (DCFLs) [8]. Within this class, the LR(0) grammars define the DCFLs with the prefix property (for any string w in a DCFL L with the prefix property, there does not exist a proper prefix of w in L: note that for any DCFL L, adding an end-marker # to L gives the prefix property). Adding more than one symbol of lookahead does not add to the class of languages defined, that is the LR(1) grammars generate the DCFLs, but for any k there are grammars which can be parsed with k symbols of lookahead but not $k-1$. The second reason for investigating LR parsers is that they are practical tools with time complexity $O(n)$ which are used in production programming environments. Thirdly, there are parser-generators which take a grammar, possibly ambiguous, and produce an LR parser for the language generated; examples are the parser-generator used in the Helsinki system HLPA [11] and the UNIX tools yacc [7] and bison [3]. We assume familiarity with LR(k) parsing as presented in for example [1].

Minimum distance

The *minimum distance* is defined for a pair of strings over a finite alphabet and a set of transformations on the alphabet, for example the familiar edit operations of replacement, insertion and deletion. The minimum distance is defined to be the minimum number of transformations required to translate one string into the other. For a finite alphabet Σ and a set of transformations

$$\Delta = \{\ (a, b) \mid a, b \in \Sigma \cup \{\lambda\}, (a, b) \neq (\lambda, \lambda)\ \}$$

we write

$$u \rightarrow v \ \mathit{via}\ a \rightarrow b \text{ in } \Delta$$

if $(a, b) \in \Delta$ and there are strings w, x in Σ^* where $u = wax$ and $v = wbx$. For a sequence of transformations $T = t_1 t_2 ... t_n$, $t_i = (a, b) \in \Delta$, we write $u \rightarrow v\ \mathit{via}\ T$ if there are strings $w_1, \ldots, w_{n-1}$ in Σ *such that

$$u \rightarrow w_1 \ \mathit{via}\ t_{1,}$$

$$w_i \rightarrow w_{i+1} \ via \ t_{i+1} \text{ for } i = 1, \ldots, n-2,$$

$$w_{n-1} \rightarrow v \ via \ t_n .$$

For strings u, v in $\Sigma *$, the minimum distance $d(u, v)$ is given by

$$d(u, v) = \min \{ \ n \mid u \rightarrow v \ via \ S \text{ for some sequence } S = s_1 ... s_n \ \}.$$

This notion may be extended to include a measure of the cost of the translation, by associating a cost with each transformation and summing the costs to provide a minimum distance cost. Let $\gamma : \Delta \rightarrow \mathbf{N}$ be a cost function and define the cost of a sequence of transformations in the natural way: for $T = t_1 t_2 ... t_n$, $t_i \in \Delta$,

$$\gamma(T) = \sum_{i=1}^{n} \gamma(t_i).$$

The minimum distance cost $\delta(u, v)$ is given by

$$\delta(u, v) = \min \{ \ \gamma(T) \mid u \rightarrow v \ via \ T \ \}.$$

A global minimum distance error correcting parser for a context-free grammar $G = (N, \Sigma, P, S)$ takes any x in $\Sigma *$ and produces a parse for some w in $L(G)$ such that the minimum distance between w and x is as small as possible. Thus a global correction scheme does not handle each error in the input separately, but finds a best approximation to incorrect input over the whole input string. Such a scheme is not held to be practical: "Unfortunately, global correction techniques cannot be incorporated in practical parsers without drastically affecting the parsing speed even of correct sentences. In fact, the global correction approach seems to fit conveniently only general context- free parsing algorithms" [12]. Aho and Peterson [2] present an $O(n^3)$ algorithm for minimum distance error correction for any context-free grammar G. They add a set of error productions to give a covering grammar G' which generates $\Sigma *$, and design a parser for G' that uses as few error productions as possible in parsing any string w. The error productions used in a parse indicate the errors in w. G' is ambiguous and the parser is a variation of Earley's algorithm [5] which includes a count of the number of error productions used. An alternative approach is used by Mauney and Fischer [10] to achieve what they term "regional least-cost repair", i.e. minimum distance error correction over a bounded region of the input string. They develop a global least-cost error correcting parser and then restrict it to regional least-cost. Instead of extending the original grammar with error productions, the parser is modified so that it can simulate insertions, deletions and replacements on the input string. For the general context-free parsing method required, they choose the Graham, Harrison, Ruzzo algorithm [6]. This method, which is similar to Earley, finds the parse in time $O(n^3)$. To achieve a more practical method, they propose using an $O(n)$ parser and calling the repair algorithm only when needed, to parse a substring of the input. An algorithm is needed to output a string which can follow the input accepted so far, for example a context-free grammar which describes the legal suffixes. This however is not presented.

Our aim is to design a parser which is practical in time and space and incorporates global minimum distance error correction, with no penalty for parsing correct input. We use the LR parsing method together with an error recovery technique which can be automated, thus permitting it to be built in to an LR parser generator such as *yacc* [7]. A technique which performs well for single token errors has already been implemented and used with success as a tool for constructing various compilers [4]. Global error correction should improve on this technique for repairs of multiple errors and complex errors.

Overview of the Method

Each state of an LR parser contains information on all the possible moves from that state: which input symbols can be shifted and which reductions are possible (LR(1) parsers inspect one symbol of lookahead to determine whether a reduction is legal). The parser detects an error in the input when there is no legal move (neither shift nor reduce) on the current configuration, and invokes the recovery method. The method generates *legal continuation strings* for the input already parsed, i.e. prefixes of legal suffixes of the parsed input; the actual remaining input is not inspected during generation of the continuation strings. One of the continuation strings is then chosen as the repair to the actual input; the minimum distance measure from the actual input is used to choose the best of the continuation strings.

The continuation strings are formed from successive legal shift moves of the parser. Each possible shift move will cause an input symbol to be concatenated to a continuation string. The method constructs *recovery configurations* of the parser which consist of a parser stack and a continuation string. The initial recovery configuration is formed from the parser stack at the point of detection of error and the empty string. For each recovery configuration, all legal moves from the top state of the stack are considered. A shift move gives rise to a new recovery configuration consisting of the stack with the shift state pushed on and the previous configuration's continuation string with the shift symbol concatenated. A reduce move gives rise to a new recovery configuration consisting of the reduced stack and the previous continuation string (there is no inspection of a lookahead symbol).

The LR Parsing Machine with Minimum Distance Error Recovery

A context-free grammar G is represented by a tuple (N, Σ, P, S) where

N is a finite alphabet of non-terminals

Σ is a finite alphabet of terminals

P is a finite set of productions P: $N \rightarrow (N \cup \Sigma)*$

Sin N is the start symbol.

An LR(0) parsing automaton M for an LR(0) grammar G may be represented by a tuple (Q, Σ, q_0, δ, γ) where

Q is a finite set of states (the stack alphabet)

q_0 in Q is the start state

Σ is the finite input alphabet (terminals of G)

δ is the ACTION transition function

$$\delta\colon Q \times \{\Sigma \cup \lambda\} \to SHIFT \times Q \cup REDUCE \times P \cup ACCEPT \cup ERROR$$

γ is the GOTO transition function $\gamma\colon Q \to Q$

Moves of the parsing automaton are either shift moves which consume one input symbol and push a state onto the stack, or reduce moves which consume no input and replace zero or more of the top states of the stack with a new state, or halting moves which accept or announce error. A configuration of the parser stack and input is represented by an instantaneous description (ID) of the form $[\ q_0 \ldots q_m,\ a_j \ldots a_{j+k}\ ]$ where the q_i represent the stack (q_m the top state) and the a_i represent the remaining input. The symbol $\vdash$ denotes the relation "move in one step" on IDs. A shift move is then denoted by successive IDs

$$[\ q_0 \ldots q_m,\ a_j a_{j+1} \ldots a_{j+k}\ ] \vdash [\ q_0 \ldots q_m q',\ a_{j+1} \ldots a_{j+k}\ ]$$

where $\delta(\ q_m,\ a_j) = (\ SHIFT,\ q'\ )$

A reduce move is denoted by successive IDs

$$[\ q_0 \ldots q_{m-n} \ldots q_m,\ a_j \ldots a_{j+k}\ ] \vdash [\ q_0 \ldots q_{m-n} q',\ a_j \ldots a_{j+k}\ ]$$

where $\delta(\ q_m,\ \lambda) = (\ REDUCE,\ A \to \alpha\ ), |\alpha| = n$, and $\gamma(\ q_{m-n}\ ) = q'$

A recovery configuration of the parser is represented by an ID $\mathbf{[}\ q_0 \ldots q_m,\ u\ \mathbf{]}$, $u \in \Sigma^*$, where the q_i represent the stack as before and u represents the continuation string (we use $\mathbf{[}$ and $\mathbf{]}$ in place of [and] to distinguish IDs representing recovery configurations from IDs representing ordinary configurations). The relation "move in one step" on recovery configurations, denoted by the symbol $\vdash_R$, is defined analogously to $\vdash$ on parser configurations, as follows.

If $\delta(\ q_m,\ a\ ) = (\ SHIFT,\ q'\ )$, $a \in \Sigma$, then $\mathbf{[}\ q_0 \ldots q_m,\ u\ \mathbf{]} \vdash_R \mathbf{[}\ q_0 \ldots q_m q',\ ua\ \mathbf{]}$

If $\delta(\ q_m,\ \lambda\ ) = (\ REDUCE,\ A \to \alpha\ )$, $|\alpha| = n$, and $\gamma(\ q_{m-n}\ ) = q'$,

then $\mathbf{[}\ q_0 \ldots q_m,\ u\ \mathbf{]} \vdash_R \mathbf{[}\ q_0 \ldots q_{m-n} q',\ u\ \mathbf{]}$

Let $\vdash_R^*$ denote the reflexive and transitive closure of $\vdash_R$. Let the configuration of the parser when error is detected be denoted by ID $[\ q_0 \ldots q_e,\ a_j \ldots a_{j+k}\ ]$. Then the set Θ_L of continuation strings of length L is given by

$$\Theta_L = \{\ u \mid u \in \Sigma^L,\ \mathbf{[}\ q_0 \ldots q_e,\ \lambda\ \mathbf{]} \vdash_R^* \mathbf{[}\ q_0 \ldots q_n,\ u\ \mathbf{]}\ \}$$

Generation of the Continuation Strings

We give a design in Pascal for the algorithm to generate the continuation strings. First, we outline the data types to be used.

TOKEN represents an input symbol (terminal or lexical token) of the grammar.
PRODUCTION represents a production of the grammar.
STRING abstracts a string of tokens with operations *print, concat, length.*
TOKENSET abstracts a set of tokens with iterator *nextT.*
PRODUCTIONSET abstracts a set of productions with iterator *nextP.*

STACK abstracts a stack of (LR(1)) parser states with operations

shift: STACK $\times$ TOKEN $\rightarrow$ STACK
reduce: STACK $\times$ PRODUCTION $\rightarrow$ STACK
LegalShifts: STACK $\rightarrow$ TOKENSET
LegalReductions: STACK $\rightarrow$ PRODUCTIONSET
GenRepairs: STACK $\times$ STRING $\rightarrow$ { }

The operations LegalShifts and LegalReductions inspect the top state of the stack to return the tokens which it is possible to shift and the productions by which it is possible to reduce respectively.

The algorithm is implemented by the recursive procedure *GenRepairs*. The input to the algorithm is the current configuration of the parser, consisting of the current stack, passed as a parameter, and the actual remaining input to the parser. The output is a sequence of strings of tokens (the continuation strings).

```
procedure GenRepairs(stack: STACK, continuation: STRING);
const REPAIRLENGTH = 10;    { desired length of continuation }
var shifts: TOKENSET;    { legal shifts from current state }
    t: TOKEN;    { next symbol for shift move }
    reductions: PRODUCTIONSET;    { legal reductions from current state }
    p: PRODUCTION;    { next production for reduce move }
begin
    if length(continuation) >= REPAIRLENGTH
        then print ( continuation )
        else begin
            { compute legal shifts }
            shifts := LegalShifts(stack);
            t := nextT(shifts);
            while t <> 0 do
                begin
                    GenRepairs(shift(stack, t), concat(continuation, t));
                    t := nextT(shifts)
                end;
            { compute legal reductions }
            reductions := LegalReductions(stack);
            p := nextP(reductions);
            while p <> 0 do
                begin
                    GenRepairs(reduce(stack, p), continuation);
```

```
                p := nextP(reductions)
            end
        end
end;
```

Computing the minimum distance

We use the Fischer-Wagner algorithm [13] to compute the edit distance of two strings u, v of lengths m and n over a finite alphabet, with error transformations insert, delete and replace. This algorithm constructs an $m \times n$ matrix M where $M(i, j)$ gives the minimum cost of transforming the prefix of length i of string u into the prefix of length j of string v. The algorithm has time complexity $O(mn)$ (Masek and Paterson [9] improve this algorithm to one of $O(n \,.\, max(1,\ m/log\ n))$ by use of the Four Russians' idea).

Restarting the parser

In order to continue parsing the remaining input after having formulated a good repair, it is desirable for the parser to be synchronized with the input. We wish to identify the most appropriate continuation string with which to replace a portion of the input. If continuation strings of length L are generated, then the minimum distances of the continuation strings from a prefix of length $2L$ of the upcoming input is computed. The continuation string with the smallest entry in the last row of the Fischer-Wagner matrices is found; if the column number of this entry is c then the next c tokens on the input are replaced with the chosen continuation string. We choose the smallest entry in the last row, over all the Fischer-Wagner matrices, rather than the smallest entry in the last column of the last row, because it is possible to have a continuation string whose last Fischer-Wagner entry $M(L, 2L)$ is greater than that of another, less suitable, continuation. There may be more than one continuation string with the same smallest entry x in the last row and in this case we choose the continuation which will replace most of the upcoming input, i.e. the entry with the largest column number c. Consider for example the Fischer-Wagner matrices for actual input *abcdef*, $L = 3$, repairs *gcd*, *gab* :

	λ	a	b	c	d	e	f
λ	0	1	2	3	4	5	6
g	1	1	2	3	4	5	6
c	2	2	2	2	3	4	5
d	3	3	3	2	2	3	4
λ	0	1	2	3	4	5	6
g	1	1	2	3	4	5	6
a	2	1	2	3	4	5	6
b	3	2	1	2	3	4	5

Here we choose the repair *gab* in preference to *gcd*, because the former allows the possibility of completing the repaired input with the string commencing *cdef*, giving an overall repair of a single insertion, whereas the latter forces at least a deletion and a replacement.

If we denote the Fischer-Wagner matrix of a continuation string u by M_u, then the required repair w in Θ_L satisfies

If $M_w(L, c) = \min \{ M_w(L, i) \mid i = 1, \ldots, 2L \}$

then for all u in Θ_L, $M_u(L, i) \geq M_w(L, c)$ for $1 \leq i \leq 2L$,

and if $M_u(L, i) = M_w(L, c)$ then $i \leq c$.

If an error configuration of the parser $[\, q_0 \ldots q_e,\, a_j \ldots a_{j+k} \,]$ gives rise to the choice of repair w,

$$w \in \Theta_L,\ [\, q_0 \ldots q_e,\, \lambda \,] \mid\!-_R^* [\, q_0 \ldots q_n,\, w \,],$$

then the parser is restarted in configuration $[\, q_0 \ldots q_n,\, a_{j+c} \ldots a_{j+k} \,]$ where c satisfies

$$M_w(L, c) = \min \{ M_w(L, i) \mid i = 1, \ldots, 2L \}$$

Pruning the Search Tree

An upper bound on the number of possible continuation strings is $|\Sigma|^L$ where L is the length of repair desired. The configurations generated by the method can be viewed as a tree with the state in which error is detected and the empty contination string at the root. We wish to reduce the size of this search tree. The first idea for pruning the tree is to use a depth-first search of the tree and to record a current best repair against which other potential repairs are measured and possibly discarded early. One configuration is followed to completion and its minimum distance from the actual input is recorded. This is the current best so far. Generation of the next configuration is started, simultaneously constructing the continuation string's Fischer-Wagner matrix. If at any stage all the entries in the current row of the matrix are greater than or equal to the current best, then this branch of the tree can be pruned. If the generation continues to completion, and its minimum distance is less than the current best, then this repair becomes the new best.

The second idea is to refine this method by estimating a likely best for the first continuation string. One way of estimating such a string is to mark the current position in the input, choose a shift or reduce on a symbol which appears, nearest, in the input, mark this new position in the input and repeat. This estimation simulates deletions; a simulation of insertions and replacements is needed. The third idea simulates these operations in a breadth-first search of the tree. At the first level of the tree, all possible configurations are generated. At the second level, we choose only those configurations whose accessing symbol (shift or reduce) appears nearest in the input; this simulates an insertion followed by zero or more deletions. This gives the following heuristic: (i) always choose the accessing symbol which appears next in the input if possible; (ii) if there is none such, simulate a single insertion followed by zero or more deletions as above. A multiple insertion can be simulated by increasing the number of levels of the search tree at which all configurations are generated from one level to k levels, for an insertion of length k symbols.

Concluding Remark

There remains to be carried out work on implementing the scheme presented here in a parser-generator and comparing the error recovery of parsers generated with that of existing parsers.

References

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, principles, techniques, and tools.* Addison-Wesley, Reading, Mass., 1986.
2. AHO, A. V., AND PETERSON, T. G. A minimum-distance error correcting parser for context-free languages. *SIAM J. Comput. 1,* 4 (Dec. 1972), 305-312.
3. CORBETT, R. P. Static semantics and compiler error recovery. Report No. UCB/CSD 85/251, Computer Science Division (EECS), Univ. California, Berkeley, Calif., 1985.
4. DAIN, J. A. Error recovery for Yacc parsers. Computer Science Report No. 73, Dept. Computer Science, Univ. Warwick, Coventry, 1985.
5. EARLEY, J. An efficient context-free parsing algorithm. *Commun. ACM 13,* 2 (Feb. 1970), 94-102.
6. GRAHAM, S. L., HARRISON, M. A., AND RUZZO, W. L. An improved context-free recognizer. *ACM Trans. Program. Lang. Syst. 2,* 3 (July 1980), 415-462.
7. JOHNSON, S. C. Yacc - yet another compiler-compiler. Computing Science Technical Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J., 1978.
8. KNUTH, D. E. On the translation of languages from left to right. *Inf. Control 8* (June 1965), 607-639.
9. MASEK, W. J., AND PATERSON, M. S. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci. 20* (Feb. 1980), 18-31.
10. MAUNEY J., AND FISCHER, C.N. A forward move algorithm for LL and LR Parsers. *ACM SIGPLAN Notices 17,* 6 (June 1982), 79-87.
11. RAIHA, K-J., SAARINEN, M., SARJAKOSKI, M., SIPPU, S., SOISALON-SOININEN, E., AND TIENARI, M. Revised report on the compiler writing system HLP78. Report A-1983-1, Dept. Computer Science, Univ. Helsinki, Finland, 1983.
12. SIPPU, S. Syntax Error Handling in Compilers. Report A-1981-1, Dept. Computer Science, Univ. Helsinki, Finland, 1981.
13. WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *J. ACM 21,* 1 (Jan. 1974), 168-173.