Information Processing Letters 23 (1986) 311–316
North-Holland

3 December 1986

EFFICIENT COMPUTATION OF THE LOCALLY LEAST-COST INSERTION STRING FOR THE LR ERROR REPAIR

Kwang-Moo CHOE and Chun-Hyon CHANG

Department of Computer Science, Korea Advanced Institute of Science and Technology, P.O. Box 150, Chongyangni, Seoul 131, Republic of Korea

Communicated by H.R. Wiehle
Received 11 November 1984
Revised 9 July 1985 and 12 December 1985

Keywords: Locally least-cost LR error repair, least-cost insertion string, expected vocabulary string

1. Introduction

The locally least-cost error repair scheme, originally proposed by Fischer et al. [5,6], is an error repair scheme which allows the construction of a table-driven or algorithmic error repairing parser with the insertion and deletion cost of the terminal symbols given by the compiler designers.

The principal idea governing locally least-cost error repair is that an input symbol at an error-detection point may always be edited to some string which allows the parser to continue normally [3]. The deletion is performed by comparing the insertion cost of the least-cost insertion string for the error symbol and the deletion cost of the error symbol [5]. The major problem in the locally least-cost error repair is to find out the least-cost insertion terminal string to be inserted. Fischer et al. [6] presented a novel scheme and the corresponding algorithm for computing the least-cost insertion string for LL(1) parsing. Anderson and Backhouse [2] presented an alternative method. But, the formalism and algorithm are rather inefficient for the LR-based parsing—Fischer et al.'s scheme must have the *closure graph* for all states, and the computing formula is not expressed explicitly [5].

In this paper, the explicit formalisms and efficient algorithms for the LR-based locally least-cost insertion string are presented. They are based on the new LALR formalisms [4,7] recently proposed by the present authors, which lead to more efficient handling of CLOSURE [1].

2. Locally least-cost error repair for LR-based parsing

In this section, the basic terminology and the definitions for LR-based parsing to be used in this paper are introduced, and the locally least-cost insertion error repair for LR-based parsing proposed by Fischer et al. [5] is described.

A context-free grammar (CFG, for short) G is a quadruple $G = (N, T, P, S)$, where N is a finite set of nonterminal symbols, T is a finite set of terminal symbols such that $N \cap T = \emptyset$; P is a finite subset of $N \times V^*$, where V (vocabulary) stands for $N \cup T$, and each member (A, α) is called a production, written $A \rightarrow \alpha$; and S is a start symbol in N. Given a CFG $G = (N, T, P, S)$, the corresponding augmented grammar is $G' = (N', T, P', S')$, where S' is a new start symbol not in N, $N' = N \cup \{S'\}$, and $P' = P \cup \{S' \rightarrow S\}$.

Familiarity with the terminology and definitions [1] related to context-free grammars and LR-based parsing is assumed. Greek letters such as α, β, γ, and ω denote vocabulary strings in V^*;

lower case Roman letters at the beginning of the alphabet, e.g., a, b, and c, are terminals in T, whereas those near the end, e.g., x, y, and z, are terminal strings in T*; upper case Roman letters at the beginning of the alphabet, e.g., A, B, and C, are nonterminals, whereas those near the end, e.g., X, Y, and Z, are vocabulary symbols in V. The length of the string α is denoted as $|\alpha|$. The empty string is denoted as ε and is such that $|\varepsilon| = 0$.

An LR(0) state is a set of LR(0) items of the form $[A \to \alpha_1 \cdot \alpha_2]$ where $A \to \alpha_1 \cdot \alpha_2$ is a production. The LR(0) item $[A \to \alpha_1 \cdot \alpha_2]$ is said to be a *kernel* item if $\alpha_1 \neq \varepsilon$ or $A = S'$, and a *closure* item otherwise.

Let the left context in the LR parsing algorithm, denoted by σ_n, be a sequence of parsing states $S_1 \ldots S_n$ in the parsing stack, where n indexes the top of the stack. The parsing configuration, denoted as (σ_n, x), is defined as the pair of the left context and the remaining input string. The **shift** and **reduce** actions in LR parsing transform the parsing configuration as follows, where ACTION and GOTO are described by Aho and Ullman [1].

Let the parsing configuration be $(\sigma_n, ax\$)$.

(1) If $\text{ACTION}[S_n, a] =$ **shift** q, then

$$(\sigma_n, ax\$) \vdash (\sigma_{n+1}, x\$), \quad \text{where } S_{n+1} = q.$$

(2) If $\text{ACTION}[S_n, a] =$ **reduce** $A \to \alpha$, then

$$(\sigma_n, ax\$) \vdash (\sigma_{n-|\alpha|+1}, ax\$),$$

$$\text{where } S_{n-|\alpha|+1} = \text{GOTO}(S_{n-|\alpha|}, A).$$

The endmarker \$ is assumed to be not in T.

It is well known that canonical LR(1) parsers have the immediate error detection property, that is, they can detect an error upon first encountering an erroneous input symbol. However, since the LALR and SLR parsing algorithms do not have the immediate error detection property, a **reduce** stack should be maintained, which stores the **reduced** states, and is cleared when the terminal **shift** action occurs. When an error is detected, the reduced stack should be used to reinstate the left context to the point at which the error symbol was first used [5].

Definition 2.1 (*insertion and deletion cost*). IC(a) and DC(a) denote the positive integer value of the insertion and deletion costs of the terminal symbol a, respectively. The insertion cost of the terminal string $x = a_1 \ldots a_n$ is defined as the sum of the insertion costs of the terminal symbols contained, i.e.,

$$IC(x) = IC(a_1) + \cdots + IC(a_n).$$

Definition 2.2 (*locally least-cost insertion string for the left context and the error symbol*, $LCI(\sigma_n, a)$). Let σ_n be a left context, and a be an error symbol. Then, the locally least-cost insertion string $LCI(\sigma_n, a)$ is defined as follows:

$$LCI(\sigma_n, a) = \begin{cases} x & \text{if } IC(x) \leqslant IC(y) \text{ for all y such that} \\ & (\sigma_n, yaz\$) \vdash^+ (\sigma_m, z\$) \text{ and} \\ & (\sigma_n, xaz\$) \vdash^+ (\sigma_{m'}, z\$) \\ & \text{for some } \sigma_m, \sigma_{m'}, \text{ and z;} \\ ? & \text{if there does not exist an x such that} \\ & (\sigma_n, xaz\$) \vdash^+ (\sigma_i, z\$), \text{ where } ? \notin T \\ & \text{and } IC(?) = \infty. \end{cases}$$

The error repair parser can resume parsing until the error symbol is **shift**-ed by inserting the least-cost insertion string x, provided that it is not "?". If the insertion cost of the least-cost insertion string x is greater than the deletion cost of the error symbol a, the error symbol becomes a candidate for deletion. If the insertion cost of x is still greater than the sum of the deletion cost of a, and the insertion cost for the same left context σ_n and the next input symbol b ($z = bz'$), the error symbol a is deleted and the new least-cost insertion string $LCI(\sigma_n, b)$ becomes a new candidate for the insertion. These procedures are repeated until the insertion cost of the current candidate for insertion is less than or equal to the sum of the deletion cost of the current input symbol and the insertion cost of the next candidate for insertion (with the current input symbol deleted).

The following algorithm characterizes this error repair scheme.

Algorithm Locally Least-Cost Error Repair

```
( * assume a_1 ... a_m denotes the input string and
   a_m = $ * )
var i : integer; x, y : terminal_string;
begin
   i := 1;
   x := LCI(σ_n, a_1);
   y := LCI(σ_n, a_2);
   while IC(x) > DC(a_i) + IC(y) do
      delete a_i;
      i := i + 1;
      x := y; y := LCI(σ_n, a_{i+1})
      od;
   insert x
end.
```

The deletion cost of the endmarker \$ should be infinite, so that the **while**-loop in the above algorithm terminates properly ($a_m = \$$). Furthermore, the above error repair scheme always terminates with the deletion and/or insertion of a terminal string without including the endmarker \$ nor "?", since $LCI(\sigma_n, \$)$ is not "?" for any left context σ_n.

The computation of the $LCI(\sigma_n, a)$ is the main part of this error repair scheme. The explicit formula and an efficient algorithm for the computation of the insertion string are derived in the following sections.

3. Formula for least-cost insertion string

In this section, an explicit and efficient formula for the least-cost insertion string for the left context and the error symbol are derived with the new LALR formalism [7]. The least-cost insertion string is the least-cost expected terminal string, which is derived by the vocabulary string to appear for the given left context. The set of expected vocabulary strings is defined as follows.

Definition 3.1 (*expected vocabulary string for the left context (and item)*). Let $[A \to \alpha_1 \cdot \alpha_2] \in S_n$. Then, the expected vocabulary string for the left context σ_n is

$$EVC(\sigma_n) = \bigcup_{[A \to \alpha_1 \cdot \alpha_2] \in \text{KERNEL}(S_n)} \{\alpha_2 \cdot EVCI(\sigma_n, [A \to \alpha_1 \cdot \alpha_2])\},$$

where KERNEL(S_n) denotes the set of *kernel* items in the state S_n, and the expected vocabulary string for the left context σ_n and the item $[A \to \alpha_1 \cdot \alpha_2]$ is

$$EVCI(\sigma_n, [A \to \alpha_1 \cdot \alpha_2]) = \bigcup_{[B \to \beta_1 \cdot A\beta_2] \in S_{n-|\alpha_1|}} \{\beta_2 \cdot EVCI(\sigma_{n-|\alpha_1|}, [B \to \beta_1 \cdot A\beta_2])\},$$

where $EVCI(\sigma_1, [S' \to .S]) = \{\$\}$.

EVC is simply the union of concatenations of vocabulary string α_2 and EVCI. EVCI is computed recursively until the left context contains the initial state only. The formula for EVCI can be rewritten using the new LALR formalism [7], which factors out the CLOSURE [1] relations from the states to the nonterminals and results in efficient handling of the CLOSURE relations.

Definition 3.2 (*L-relation and L-graph* [7]). Let G be a context-free grammar (CFG). Then, the L-graph is a directed graph whose vertices are the same as the nonterminal symbols in G, and the edges are

$$Edge(A, B) = \{\beta \mid A \to B\beta \in P\}.$$

The nonterminal A has an L-relation to B, written A L B, iff $Edge(A, B) \neq \emptyset$.

Theorem 3.3

$$EVCI(\sigma_n, [A \to \alpha_1 \cdot \alpha_2]) = \bigcup_{\substack{A' \, L^* \, A \\ [C \to \gamma_1 \cdot A'\gamma_2] \in \text{KERNEL}(S_{n-|\alpha_1|})}} \{PATH(A', A) \cdot \gamma_2 \cdot EVCI(\sigma_{n-|\alpha_1|}, [C \to \gamma_1 \cdot A'\gamma_2])\},$$

with the notation

$$PATH(A', A) = \bigcup_{\text{paths}} \{\omega_n \ldots \omega_1 \mid B_0 = A', B_n = A, n \geqslant 0, B_0 \to B_1\omega_1 \in P, \ldots, B_{n-1} \to B_n\omega_n \in P\},$$

where the sequence $\omega_1 \ldots \omega_n$ *describes a path* A′ *to* A *in the* L*-graph of the form*

$$\begin{array}{ccccccc} (B_0) & \xrightarrow[\omega_1]{} & (B_1) & \xrightarrow[\omega_2]{} & \cdots & \xrightarrow[\omega_n]{} & (B_n) \cdot \\ \| & & & & & & \| \\ A' & & & & & & A \end{array}$$

The new formula EVCI considers the *kernel* item $[C \to \gamma_1 \cdot A'\gamma_2]$ with $A'\ L^*\ A$ in the state $S_{n-|\alpha_1|}$, whereas the old formula considers the item $[B \to \beta_1 \cdot A\beta_2]$ in the same state. Note that the old formula should be repeatedly used in the same state $S_{n-|\alpha_1|}$ until the *kernel* item $[C \to \gamma_1 \cdot A'\gamma_2]$ is found if the item $[B \to \beta_1 \cdot A\beta_2]$ is a *closure* item. PATH(A′, A) in the new formula prevents the recursive use of the formula in the *same* state. PATH is a relation between nonterminals, whereas CLOSURE is a relation between states. The new formula factors out the relation (PATH) which is dependent on nonterminals only from the state (CLOSURE), which leads to the efficient computing of the locally least-cost insertion string.

The least-cost prefix string for the given set of vocabulary strings and the error symbol is defined so as to describe the least-cost insertion string for the given left context and the error symbol.

Definition 3.4 (*least-cost prefix string for the vocabulary string and the error symbol* (LCE)). Let L be a set of vocabulary strings, and let a be an error symbol. Then, the least-cost terminal string preceding the error symbol a, which is derivable from the vocabulary string set L, is defined as follows:

$$\mathrm{LCE}(L, a) = \begin{cases} x & \text{if } \mathrm{IC}(x) \leqslant \mathrm{IC}(y) \text{ for all } \alpha \in L \text{ such that} \\ & \alpha \Rightarrow^* yaz \text{ and } \beta \Rightarrow^* xaz' \\ & \text{for some } \beta \in L, z \text{ and } z'; \\ ? & \text{if there does not exist an } \alpha \in L \\ & \text{such that } \alpha \Rightarrow^* yaz. \end{cases}$$

Corollary 3.5 (least-cost insertion string). *Let* σ_n *be a left context and let* a *be an error symbol. Then*

$$\mathrm{LCI}(\sigma_n, a) = \mathrm{LCE}(\mathrm{EVC}(\sigma_n), a).$$

It is to be observed that, even when cycles are present in the L-graph, only paths in PATH, which never traverse the cycle more than once, are necessary in computing LCI due to the definition of LCE, and that the total number of paths involved is finite.

4. Computing locally least-cost insertion string

Having characterized the locally least-cost insertion string in the previous section, in this section the efficient algorithm for computing the least-cost insertion string is described.

Definition 4.1 (*least-cost derivable terminal string* (LCD)). Let L be a set of vocabulary strings. Then, the least-cost derivable terminal string of the vocabulary string set L is defined as follows:

$$\mathrm{LCD}(L) = x \quad \text{if } \mathrm{IC}(x) \leqslant \mathrm{IC}(y) \text{ for all } \alpha \in L \text{ such that } \alpha \Rightarrow^* y \text{ and } \beta \Rightarrow^* x \text{ for some } \beta \in L.$$

The computation of the least-cost insertion string is to find out the least-cost prefix string for the concatenation and union of the expected vocabulary strings. The following algorithm, which computes the least-cost insertion string for the given left context and error symbol, is derived from the previous definitions, corollary, and theorems.

Algorithm Compute_Locally_Least_Cost_Insertion_String

```
Input parameters:
Stack : array[Stack_Range] of State;
  ( * Parsing Stack * )
Top : Stack_Range;  ( * Stack Pointer * )
a : Terminal;  ( * Error Symbol * )

function LCI : T_D_String;
  var
    x : T_D_String;
    y : T_String;
  function LCD(α : V_String) : T_String;
  function LCE(α : V_String; a : Terminal)
           : T_D_String;
```

```
Function Min_IS(α : V_String; var y : T_String)
        : boolean;
begin  ( * Min_IS * )
  if IC(x) > IC(y) + IC(LCE(α, a))
    then x := y · LCE(α, a) fi;
  if IC(x) > IC(y) + IC(LCD(α))
    then y := y · LCD(α); return true
    else return false
  fi
end;  ( * Min_IS * )

procedure Back(SP: Stack_Range; [A → α1 · α2]
                : Item; y : T_String);
  var
    y' : T_String;
begin  ( * Back * )
  for [C → γ1 · A'γ2] ∈ KERNEL(Stack[SP]) and
      A' L* A do
      y' := y;
    if Min_IS(PATH(A', A) · γ2, y')
      then Back(SP − | γ1 |, [C → γ1 · A'γ2], y')
    fi
  od
end;  ( * Back * )

begin  ( * LCI * )
x := "?";
for [A → α1 · α2] ∈ KERNEL(Stack[Top]) do
  y := ε;
  if Min_IS(α2, y)
    then Back(Top − | α1 |, [A → α1 · α2], y) fi
od;
return x
end  ( * LCI * ).
```

The function LCI manages two important terminal string variables, x and y, whose types are T_D_String (terminal string and "?"), and T_String (terminal string), respectively. The variable x is the least-cost *insertion* string for the error symbol a and *all* expected vocabulary strings *examined*, and y is the least-cost *derivable* string for the *examining* vocabulary string. x is the return value of the outermost block function LCI and y is the *variable* parameter of the function Min_IS. As shown in the first **if**-statement of the function Min_IS, x is the current least-cost insertion string for the error symbol a by concatenating the current y and LCE(α, a) if the insertion cost of the current x is greater than that of the new x. The guarded command of the second **if**-statement in the function Min_IS tests the insertion cost of the new x and that of the new y, which is the concatenation of the old y and LCD(α). If the cost of the new x is greater than that of the new y, the path for y should be examined more, by returning **true**, and returning **false** otherwise.

The driver of the function LCI initializes the least-cost *insertion* string x to "?" whose insertion cost is infinite, and computes the

$$\mathrm{LCE}\left(\alpha_2 \cdot \mathrm{EVCI}\left(\sigma_{p-|\alpha_1|}, [A \to \alpha_1 \cdot \alpha_2]\right), a\right)$$

by initializing y to the empty string, checking Min_IS(α_2, y), and calling

$$\mathrm{Back}(\mathrm{TOP} - |\alpha_1|, [A \to \alpha_1 \cdot \alpha_2], y)$$

for all *kernel* items $[A \to \alpha_1 \cdot a_2]$ in Stack[TOP].

The **recursive** procedure Back computes EVCI (left context, item) by using the **recursive** calls, if the path should be examined more. The local variable y' of the **recursive** procedure Back is initialized as y in the **for**-loop so that every PATH(A', A) · γ_2 should be examined with the same y. Note that details for computing the functions LCD and LCE are omitted.

The term PATH(A', A) in the procedure Back, which inhibits the recursive calls to the same state (for CLOSURE operations), is the key to the efficiency of the new algorithm. Clearly, it is impractical to precompute and tabulate the last-cost insertion string (LCI) for all left contexts and terminal symbols.

5. Conclusion

In this paper, an explicit formula and an efficient algorithm for the locally least-cost insertion string for LR error repair using the *new* LALR formalism are presented. The formula derived is based on the idea that the string to be inserted is the least-cost terminal string derivable from the expected vocabulary string for the left context (EVC).

In Fischer et al.'s algorithm, the relations dependent only on the nonterminals are repeatedly

reexamined and restored for every item in every state (LP[S, J] in [5]). In the new algorithm proposed in this paper, the storage and/or time required for computing and storing the *closure graph* [5] for every state can be reduced to computing and storing the L-relation which is dependent on nonterminals only. The new formalism provides the efficient algorithm for the table construction and the error repair phase of compiling.

For example, in the LALR(1) grammar of Pascal [4], the total number of entries in the L-relation is 308, whereas that of the LP[S, J] is 2206. Each entry in the L-relation or LP must have least-cost terminal strings for all error symbols. The number of terminal symbols is 65. Therefore, the amount of saving on the entries for storing the terminal strings is $65 \times (2206 - 308)$, which is around 120 000.

References

[1] A.V. Aho and J.D. Ullman, Principles of Compiler Design (Addison-Wesley, Reading, MA, 1977).

[2] S.O. Anderson and R.C. Backhouse, An alternative implementation of an insertion-only recovery technique, Acta Informatica 18 (1982) 289–298.

[3] S.O. Anderson, R.C. Backhouse, E.H. Bugge and C.P. Stirling, An assessment of locally least-cost error recovery, The Comput. J. 26 (1) (1983) 15–24.

[4] K.M. Choe, A new analysis of LALR formalisms, Ph.D. Thesis, Dept. of Computer Science, Korea Advanced Institute of Science and Technology, Seoul, Korea, 1983.

[5] C.N. Fischer, B.A. Dion and J. Mauney, A locally least-cost LR-error corrector, Rept. 363, Computer Science Dept., Univ. of Wisconsin, Madison, 1979.

[6] C.N. Fischer, D.R. Milton and S.B. Quiring, Efficient LL(1) error correction and recovery using only insertions, Acta Informatica 13 (3) (1980) 141–154.

[7] J.C.H. Park, K.M. Choe and C.H. Chang, A new analysis of LALR formalisms, ACM Trans. Programm. Languages and Systems 7 (1) (1985) 159–175.