A FORMAL APPROACH TO FORWARD MOVE ALGORITHMS

by

J. Mauney
C. N. Fischer

Computer Sciences Technical Report #453

December 1981

A Formal Approach to Forward Move Algorithms

Jon Mauney

Charles N. Fischer

1. Introduction

Many algorithms have been proposed for repair of syntactic errors in a program. A number of these [2,3,4,5,11], choose the repair action by examining the next symbol in the input. In most situations, examination of this symbol reveals that there is a choice of repair actions. In such a case, the algorithm must make the choice based on a secondary criterion, such as a cost function. Given the same left context and error symbol, a one-symbol algorithm will always make the same choice, even though the remaining right context may be vastly different. Clearly, the quality of the repairs could be improved by incorporating additional right context information into the algorithm.

Several authors have proposed repair algorithms that perform a "forward move" to gather additional context. Aho and Peterson [1] presented an algorithm to find the minimum distance repair to an entire program. Their method is considered impractical since it uses Earley's parser and a much expanded grammar. Other techniques [6,8,9,10] have been more practical, but have lacked a formal model of the repairs chosen. Also, these techniques generally do not fare well when presented with a cluster of errors.

We propose a formal model of multiple-symbol repair based on the idea of a "regionally least-cost" repair, and give an algorithm that finds such repairs.

2. Regionally Least-Cost Repair

We begin by defining regionally least-cost repair. As usual, we will assume a context-free grammar, $G=(V_t,V_n,S,P)$; L(G) is the language generated by G, and Pr(G) is the set of prefixes of sentences in L(G). The problem is to repair an arbitrary string, $x \in V_t^*$, into a string, $y \in V_t^*$, that is a sentence in L(G). We will use the three primitive edit operations insert, delete, and replace, and we will search for a repair of least cost, based on cost vectors for the three operations:

IC(a) gives the cost of inserting terminal symbol a
DC(a) gives the cost of deleting terminal symbol a
RC(a,b) gives the cost of replacing a with b

We require that all costs be non-negative. We will also assume that at most one replacement or deletion is made at each input position. This assumption is equivalent to requiring that the costs satisfy "triangle inequalities":

For all $a,b,c \in V_t$

$$RC(a,b)+RC(b,c) \geq RC(a,c)$$
$$RC(a,b)+DC(b) \geq DC(a)$$
$$IC(a)+RC(a,b) \geq IC(b)$$

It is convenient to assume that RC(a,a) = 0.

Based on the cost functions, we define two other functions that extend costs to nonterminals, strings and derivations:

$C(\lambda) = 0$; (λ denotes the empty string.)

$C(a_1 \ldots a_n) = IC(a_1) + \ldots + IC(a_n)$

$C(A) = \min \{ C(x) \mid x \in V_t^*, A \stackrel{*}{\Longrightarrow} x \}$

$C(X_1 \ldots X_n) = C(X_1) + \ldots + C(X_n)$

$$\text{Derive}(A,a) = \min_{x,y \in V_t^*,\ b \in V_t} \left[\{ \infty \} \cup \{ C(xy) + RC(a,b) \mid A \stackrel{*}{\Longrightarrow} xby \} \right]$$

We are now ready to define a regionally least cost repair.

Definition: A modification, M, is a series of edit operations, $E_1E_2 \ldots E_n$, where each E_i is a insert, delete or replace operation. The string resulting from the application of the modification, M, to a string x, is written M(x). The cost of the modification, C(M(x)), is the sum of the costs of the edit operations.

Definition: Given two strings x, $y \in V_t^*$, with x in Pr(G), a repair of y following x, is a modification, M, such that xM(y) is in Pr(G).

Definition: A regionally least-cost repair of y following x is a repair, M, of y following x such that for any other such repair, N, of y following x, $C(N(x)) \geq C(M(x))$.

3. An algorithm for globally least-cost repair

We will develop the algorithm first as a globally least-cost repair algorithm. Then in the next section, we will restrict it to the more feasible regionally least-cost case.

Aho and Peterson perform their "least errors parse" by adding error productions to the grammar. We will take a complementary approach, and use a modified parser on the original grammar. An insertion can be simulated by allowing the parse state to advance over symbols that are not present in the input, a deletion by consuming a symbol of input without advancing the parse state, and a replacement by consuming one symbol while advancing the parse across a different symbol.

For our parser, we choose the algorithm of Graham, Harrison and Ruzzo [7]. This parsing algorithm produces a triangular matrix, each cell of which contains a set of "dotted productions", $A \rightarrow \alpha \cdot \beta$, representing the possible parses of a corresponding substring of the input. The dot indicates that part of the production, α, has been used to match input symbols, and that the remainder has not, yet. New cells of the matrix are created under the control of two "pasting" operators. Cells that match a longer substring of the input are created by pasting together two existing cells. The parse is advanced by pasting existing cells to the next input symbol. We introduce error repair by extending these pasting operations (and the "Predict" function), and attaching a running cost to each dotted production. When the parse is completed, we extract the repair from the matrix in much the

same way as a parse is extracted in ordinary use. The remainder of the algorithm is unchanged.

To illustrate these modifications, we will examine one of the pasting operators. According to the original definition of [7], if Q is a set of dotted rules, and a $\in V_t$ then

$$Q*a = \{ A \rightarrow \alpha B \beta \cdot \gamma \mid A \rightarrow \alpha \cdot B \beta \gamma,\ \beta \stackrel{*}{\Rightarrow} \lambda,\ \text{and}\ B \stackrel{*}{\Rightarrow} a \}$$

In other words, we can paste a dotted production and an input symbol together if the symbol immediately to the left of the dot can generate the input symbol. The dot is also moved across symbols that derive the null string, so that the next symbol can be matched in all possible positions. We introduce simulated insertions to this definition in two ways. First, the symbol to the right of the dot need not generate a single symbol, but may generate a string containing the input symbol; the remainder of that string is inserted. Second, the dot can continue to move across all the remaining symbols in the production, as if they derived the null string; the cheapest derivable terminal string is inserted. Deletions are simulated by including the original dotted production in the new set; the input symbol is consumed without advancing the parse. Replacements are made by moving the dot across a symbol, B, even though $B \stackrel{*}{\Rightarrow} a$. This possibility is handled by the Derive function, and doesn't show explicitly in the pasting operation. We will add a new component to each dotted production: a running cost of the modifications made in the parse so far. Our modified pasting operation for Q a set of dotted productions and a $\in V_t$ is:

$$Q*a = \{\ A \dashrightarrow \alpha B\beta\cdot\gamma;c \mid A \dashrightarrow \alpha\cdot B\beta\gamma;c' \in Q,\ c = c' + C(\beta) + Derive(B,a)\ \}$$
$$\cup\ \{\ A \dashrightarrow \alpha\cdot B\beta;c \mid A \dashrightarrow \alpha\cdot B\beta;c' \in Q,\ c = c' + DC(a)\ \}$$

The cost component is represented by ";c".

The effect of these extensions to the parsing algorithm is the same as the effect of adding Aho and Peterson's error productions; the sets in the parse matrix are isomorphic to those that would be obtained using the modified grammar. Therefore, the correctness and complexity of the parser should be unchanged by the modifications. The only danger is the additional cost component of the dotted productions: since the cost is potentially unbounded, the presence of dotted productions that differ only in the cost could cause the size of a set to be unbounded. However, it is easy to show that if two dotted productions in a set differ only in the cost component, then the one with higher cost can never participate in a least cost repair; any parse can be made cheaper by using the other dotted production. Therefore, a higher cost duplicate can always be discarded, and the size of the cell is not affected by the presence of cost components.

4. An algorithm for regionally least-cost repair

We have presented an algorithm that finds a least-cost repair of an entire program, in time proportional to the length of the program cubed. We do not propose that it be used as such. Instead, we intend that the repair algorithm be called only when needed, and then only to repair a reasonably sized region of the program. A linear-time parser, such as LL(1) or LR(1), can be

used for the major portion of the program.

In order to find repairs that are legal following the prefix already accepted by the parser, we will use the repair algorithm on a grammar that describes the legal continuations. Such a grammar can be derived from the state of the parser. If an LL(1) parser is used, this task is easy: the parse stack describes the expected suffix, and we replace the starting production with the production S --> stack. For an LR(1) (or SLR or LALR) parser, we can use the technique described in [3] to derive a regular expression that describes the legal suffixes. From this regular expression we can easily derive an equivalent context free grammar, which will be added to the original grammar for the purpose of repair.

Once the parse/repair algorithm has started, it can stop at any point; after each iteration of the main loop, the region of least-cost repair has been extended over one more input symbol. Thus, the algorithm can be used to find a least-cost repair over a fixed-sized region, or the size of the region can be dynamically controlled. A repair over a region of fixed size has an advantage in that it requires a fixed amount of time to compute, but the fixed size may be too small for some error situations and unnecessarily large for others. Better repairs might be found if the region size is dynamic, but in the worst case the region might include all of the remainder of the string. In practice this may not be a problem; in fact, if the expected size of a region is constant, the expected time to compute a repair may also be constant, if the distribution is reasonable. For instance, if

the size of a region is some fixed minimum, k, plus a variable part that follows an exponential distribution, $P(m)=Ce^{-Dm}$, then the expected time to repair that region is less than $C1+C2/(1-e^{-Dn-D})$, which is bounded by a constant as the size of the program, n, grows. Thus, in the average case the total time to parse and repair a program is linear in the length of the program. We are currently investigating a number of criteria for dynamically choosing a region size, and the distribution of the sizes.

After the repair has been found, control is returned to the linear-time parser. Repairs to the program can be effected in two ways. The repaired string can be physically placed into the input buffer and reparsed, or the state of the parser can be reset, using information from the parse matrix.

5. Implementation results

To illustrate the improved repairs possible with this algorithm, consider these four fragments of a Pascal program:

```
(1)  ... i := j  a := b ; ...
(2)  ... i := j  a + b ; ...
(3)  ... i := j  a[i] := b ; ...
(4)  ... i := j  a i] := b ; ...
```

Assume that all insert costs are 1, all delete costs are 4 and all replacement costs are infinite. A 1-symbol least-cost repair algorithm would make the same initial repair in (1) and (2), presumably getting the "best" repair in one of the examples, and cascaded errors in the other. A two-symbol regionally least-cost

repair would find the "best" repair in both cases, inserting a semicolon between 'j' and 'a' in (1), and inserting an operator in (2). Example (3) illustrates the problem with fixed region size. The subscript expression could be made arbitrarily long, foiling any fixed lookahead; yet to a human, the situation is almost the same as in (1). An ideal forward-move algorithm would look ahead to the ':=', but not bother to go much farther. (4) shows a cluster of errors that requires insertions in two places. Our algorithm would repair it to ... i := j; a[i] := b;...

The repair algorithm requires careful implementation if reasonable speed is expected. The effect of region size on computation time is overshadowed by the effect of grammar size, for moderate regions. Our original implementation (intentionally a "quick and dirty" approach) was not linear in the size of the grammar, and required many tens of seconds for a five symbol region on a Pascal grammar. A second version, linear in the grammar, computes a five symbol repair for Pascal in approximately three seconds (on a VAX-11/780). By using techniques similar to those suggested in [7], we can expect a more efficient implementation.

6. Conclusion

We have presented a model of error-repair using a forward move. This model provides a formal, language-level definition of how repairs are chosen, using the idea of regional least-cost. The algorithm to compute such corrections is linear in the size of the grammar, and cubic in the size of the region. Even if region size is variable, in the average case the total complexity of the parse/repair package is essentially linear.

7. References

[1] Aho, Alfred V. and Thomas G. Peterson , "A minimum distance error correcting parser for context-free languages," SIAM Journal of Computing 1, 4, pp. 305-312 (1972).

[2] Backhouse, Roland C., Syntax of Programming Languages, Theory and Practice, Prentice-Hall (1979).

[3] Fischer, Charles N., Bernard A. Dion, and Jon Mauney, "A Locally Least-Cost LR Error-Corrector," Tech. Report 363, to appear in ACM TOPLAS, University of Wisconsin-Madison (August 1979).

[4] Fischer, Charles N., Donn R. Milton, and Jon Mauney, "A locally least-cost LL(1) error corrector," Tech. Report #371, University of Wisconsin (August 1979).

[5] Fischer, Charles N., Donn R. Milton, and Sam B. Quiring,

"Efficient LL(1) error correction and recovery using only insertions," Acta Informatica 13, 2, pp. 141-154 (1980).

[6] Graham, Susan L., Charles B. Haley, and William N. Joy, "Practical LR error recovery," Sigplan Notices 14, 8, pp. 168-175 (1979).

[7] Graham, Susan L., Michael A. Harrison, and Walter L. Ruzzo, "An Improved Context-Free Recognizer," ACM Transactions on Programming Languages and Systems 2, 3, pp. 415-462 (July 1980).

[8] Graham, Susan L. and Steven P. Rhodes, "Practical syntactic error recovery," Communications of the ACM 18, pp. 639-650 (1975).

[9] Pai, Ajit B. and Richard B. Kieburtz, "Global Context Recovery: A New Strategy for Parser Recovery From Syntax Errors," ACM Transactions on Programming Languages and Systems 2, 1, pp. 18-41 (January 1980).

[10] Pennello, Thomas J. and Frank L. DeRemer, "A forward move algorithm for LR error recovery," 5th ACM Symposium on Principles of Programming Languages, pp. 241-254 (1978).

[11] Roehrich, Johannes, "Methods for the Automatic Construction of Error Correcting Parsers," Acta Informatica 13, 2, pp. 115-139 (1980).