Acta Informatica
DOI 10.1007/s00236-010-0115-6

ORIGINAL ARTICLE

LR error repair using the A* algorithm

Ik-Soon Kim · Kwangkeun Yi

Received: 20 June 2007 / Accepted: 8 February 2010

Abstract This article presents a local LR error repair method that repairs syntax errors quickly by adoption of the A* algorithm that helps remove unproductive configurations. The new method also enhances the repair quality by adoption of a flexible edit strategy to support shifting symbols unrestrictedly, as well as inserting and deleting symbols, in order to repair invalid input strings. Experimental results show that the new method excels existing works in repair quality and efficiency.

1 Introduction

The problem of repairing syntax errors during context-free parsing has received much attention [2–4,6–10,14–16,18]. The goal of error repair is to allow the parser to continue after it has encountered a syntax error so that it can identify and report subsequent errors. Error repair parsing is indispensable to syntax-directed editors when highlighting syntax errors. Error repair parsing is also indispensable to batch mode compilers when avoiding the edit-compile loop by reporting as many errors as possible in a single pass of compilation.

The parser can repair an invalid string by changing it into the closest valid string, whereas no parser can correct an invalid string in that it does not know the programmer's intention. Multiple or infinite repairs of an invalid string exist in general, so there should be an efficient way of choosing the most adequate one among multiple repairs. Error repair methods

This work was partially supported by the Engineering Research Center of Excellence of Korea Ministry of Education, Science and Technology(MEST)/National Research Foundation of Korea(NRF) Grant 2009-0063246.

I.-S. Kim (✉)
Electronics and Telecommunications Research Institute,
138 Gajeongno, Yuseong-gu, Daejeon 305-700, Korea
e-mail: ik-soon.kim@etri.re.kr

K. Yi
Seoul National University, 599 Gwanak-ro, Gwanak-gu, Seoul 151-744, Korea

Published online: 06 March 2010

adopting the least-cost repair scheme compute the total cost of inserting and deleting symbols in repairing an invalid string by associating each symbol with insertion and deletion costs, and choose the repair with the least total cost [2–4,6–10,14–16,18].

Error repair methods can be classified into "local" and "global" methods. A local repair method attempts to change the remaining input symbols when detecting a syntax error whereas a global repair method attempts to change all the input symbols including the symbols before the detected error position. This article focuses on the local error repair method.

The issue in error repair parsing is how to improve the repair efficiency and quality. For repair efficiency, we need to remove unproductive and redundant repair process. We also need to speed up the search process for repairs. For repair quality, we need to take into consideration a flexible repair strategy that allows for shifting symbols unrestrictedly, as well as inserting and deleting symbols, in order to find the most adequate error repairs for detected syntax errors.

This article presents a local LR error repair method that excels in repair efficiency and quality. Our method speeds up the repair process greatly by adopting the A* algorithm. Our method also accelerates the repair process and reduces memory use by removing unproductive and redundant configurations. Our method enhances the repair quality by adopting a flexible edit strategy that allows for unrestricted shifts, as well as insertions and deletions, of symbols. Corchuelo et al.'s method also supported shifting symbols [5], but their limited shift operation misses many useful repairs. Experimental results show that our method is excellent to repair syntax errors with respect to quality and efficiency.

This article is organized as follows. Section 2 briefly reviews the preliminaries necessary for this article. Section 3 presents an LR error repair method using the A* algorithm. Section 4 summarizes existing methods related to this article. Section 5 reports the experimental results. Section 6 concludes. The "Appendix" shows the supplementary lemmas and proofs.

2 Preliminaries

This section shows the basic terminology and definitions for this article.

2.1 LR parsing

Alphabets An alphabet is a finite nonempty set of symbols.

Strings A string is a finite sequence of symbols from the alphabet. For a string x, $|x|$ denotes the length of x, and $k : x$ denotes the prefix of x whose length is $\min\{k, |x|\}$. An empty string is written as ε.

Context free grammars A *context free grammar* 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, P is a finite subset of $N \times (N \cup T)^*$ where each member (A, α) is called a production and is written as $A \to \alpha$, and S is the start symbol in N. $N \cap T$ should be empty, and the union of the terminal and nonterminal sets, V, is called the vocabulary set. We always assume that G is a reduced grammar such that G does not contain any useless symbols. Roman letters A, B denote nonterminal symbols in N, Roman letters X, Y denote vocabulary symbols in V, Roman letters x, y, z denote terminal strings in T^*, and Greek letters $\alpha, \beta, \gamma, \delta$ denote vocabulary strings in V^*.

Augmented context free grammars An *augmented context free grammar* for $G = (N, T, P, S)$ is $G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S')$ where the new start symbol S' is not in $N \cup T$. The input string is accepted when and only when the new production $S' \rightarrow S$ is reduced.

Viable prefixes A vocabulary string γ is a viable prefix of G if a derivation relation

$$S \Rightarrow_r^* \delta Ay \Rightarrow_r \delta\alpha\beta y = \gamma\beta y$$

holds in G for some vocabulary string δ, $y \in T^*$ and production $A \rightarrow \alpha\beta$ in G. A prefix of a viable prefix is also a viable prefix.

LR(k) items A pair of the form $[A \rightarrow \alpha \bullet \beta, x]$ is an LR(k) item for $k \geq 0$ if $A \rightarrow \alpha\beta$ is a production in P and x is a string in $k : T^*$. The first part of an item $A \rightarrow \alpha \bullet \beta$ is called the core of the item, and the second part x is a lookahead string. In case of $k = 0$, lookahead strings are omitted. Kernel items are items whose core parts are either $S' \rightarrow \bullet S$ or $A \rightarrow \alpha \bullet \beta$ ($\alpha \neq \varepsilon$).

Canonical LR(k) machines The canonical LR(k) machine M is a deterministic finite automaton (DFA) for recognizing viable prefixes associated with grammar $G = (N, T, P, S)$ such that

$$M = (Q_M, V, P_M, q_s, \varnothing)$$

where

- Q_M is the set of all states of M,
- V is the union of N and T,
- P_M is the set of all transitions of M, and
- q_s is the initial state of M.

The canonical LR(k) machine M can be constructed as follows. The initial state q_s is

$$q_s = \text{closure}([S' \rightarrow \bullet S, \$^k])$$

where closure(K) is the smallest set satisfying

$$\begin{aligned}&\text{closure}(K)\\ &= K\\ &\cup \{[B \rightarrow \bullet\alpha, \text{FIRST}_k(\beta x)] \mid [A \rightarrow \alpha \bullet B\beta, x] \in \text{closure}(K), B \rightarrow \alpha \in P\}\end{aligned}$$

and $\text{FIRST}_k(\alpha) = \{k : x \mid \alpha \Rightarrow^* x, x \in T^*\}$. Initially, we let $Q_M = \{q_s\}$. For $q \in Q_M$ and $X \in V$, we compute a new state q' as

$$q' = \texttt{move}(q, X)$$

where $\texttt{move}(q, X) = \text{closure}(\{[A \rightarrow \alpha X \bullet \beta, x] \mid [A \rightarrow \alpha \bullet X\beta, x] \in q\})$, and insert q' and $q \xrightarrow{X} q'$ into Q_M and P_M, respectively. This process repeats until nothing more can be added to Q_M and P_M. Figure 1 shows the algorithm for constructing M.

Construction of the LR parsing table For an augmented grammar G', the canonical LR(1) parsing table for G', action, can be constructed as follows. Let $M = (Q_M, V, P_M, q_s, \varnothing)$ be the canonical LR(1) machine using the construction algorithm in Fig. 1, and let $Q_M = \{q_0, q_1, ..., q_n\}$. The parsing actions for q_i are determined as follows.

Fig. 1 Algorithm for constructing the canonical LR(k) machine $M = (Q_M, V, P_M, q_s, \varnothing)$ for grammar $G = (N, T, P, S)$. Q_M will contain all states of M, P_M will contain all transitions of M, and q_s will be the initial state of M [20]

$q_s := \text{closure}([S' \to \bullet S, \$^k])$
$Q_M := \{q_s\}$
$P_M := \varnothing$
repeat
  for all $q \in Q_M$ and $X \in V$ **do**
    $q' := \texttt{move}(q, X)$
    $Q_M := Q_M \cup \{q'\}$
    $P_M := P_M \cup \{q \xrightarrow{X} q'\}$
until nothing more can be added to Q_M and P_M.

- action$[q_i, a] =$ `shift` q_j for a terminal symbol a if $[A \to \alpha \bullet a\beta, b] \in q_i$ and $\texttt{move}(q_i, a) = q_j$.
- action$[q_i, A] =$ `goto` q_j for a nonterminal symbol A if $[B \to \alpha \bullet A\beta, b] \in q_i$ and $\texttt{move}(q_i, A) = q_j$.
- action$[q_i, a] =$ `reduce` $A \to \alpha$ if $[A \to \alpha\bullet, a] \in q_i$, $A \neq S'$.
- action$[q_i, \$] =$ `accept` if $[S' \to S\bullet, \$] \in q_i$.

All table entries not defined by the above rules are set to "`error`". If some shift-reduce or reduce-reduce conflicts occur during the construction of the action table, the grammar is not an LR(1) grammar.

LR parsing The LR parsing algorithm decides whether a terminal input string belongs to the language an LR context-free grammar generates. This algorithm scans the input string from left to right in a bottom-up style.

The LR(1) parsing algorithm starts from the initial state q_s, and decides the next parsing action by using action$[q, a]$ where q is the top state of parsing stack and a is the next input symbol. We assume that the input string always ends with \$. The LR(1) parsing algorithm can be expressed using transition rules `(LR-SHIFT)` and `(LR-REDUCE)` in Fig. 2 such that configuration $(q_1 \ldots q_i, a_1 \ldots a_m)$ is composed of a state sequence $q_1 \ldots q_i$ and the remaining input string $a_1 \ldots a_m$. If there is no syntax error in the input string $xy\$$, the LR transition using the transition rules will be

$$(q_s, xy\$) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_i, y\$) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j, \$)$$

where action$[q_j, \$] =$ `accept`.

False reductions in LALR parsers The tight definition of lookahead sets in an LR(k) parser ensures that reductions are not carried out once an incorrect input symbol is seen. However, LALR(k) parsers can have larger lookahead sets and may perform further reductions after an erroneous input symbol is read. Hence, in repairing syntax errors, the LALR(k) parser needs to use the reduction stack [8] to recover parsing stack after false reduction happens. When we say in this article that *an error is detected at a state sequence* $q_1 ... q_n$, the state sequence $q_1 ... q_n$ is a restored parsing stack using the reduction stack.

`(LR-SHIFT)` $$\frac{\text{action}[q_i, a_1] = \texttt{shift}\ q}{(q_1 \ldots q_i,\ a_1 \ldots a_m) \xrightarrow{\text{LR}} (q_1 \ldots q_i q,\ a_2 \ldots a_m)}$$

`(LR-REDUCE)` $$\frac{\text{action}[q_i, a_1] = \texttt{reduce}\ A \to \alpha \qquad \text{action}[q_{i-|\alpha|}, A] = \texttt{goto}\ q}{(q_1 \ldots q_i,\ a_1 \ldots a_m) \xrightarrow{\text{LR}} (q_1 \ldots q_{i-|\alpha|} q,\ a_1 \ldots a_m)}$$

Fig. 2 LR(1) transition rules

2.2 Insertion and deletion costs

Insertion cost $ic(a)$ and deletion cost $dc(a)$ denote, respectively, the positive values of the insertion and deletion costs of a terminal symbol a. The insertion and deletion costs of a terminal string $a_1 ... a_n$ are defined as follows.

$$ic(a_1 ... a_n) = ic(a_1) + \cdots + ic(a_n)$$
$$dc(a_1 ... a_n) = dc(a_1) + \cdots + dc(a_n)$$

The insertion cost is extended for a vocabulary string α as

$$ic(\alpha) = \min\{ic(x) \mid \alpha \Rightarrow^* x, x \in T^*\}.$$

The deletion cost of \$, $dc(\$)$, is defined to be infinite so that \$ may not be deleted.

Defining insertion and deletion costs is entirely the compiler designer's job. The compiler designer should devise the proper costs that will be helpful to repair syntax errors effectively with his knowledge and experience on the target language or grammar.

2.3 Graph search algorithms

We provide the basic terminology and definitions on graphs which will be necessary to develop the efficient repair-finding algorithm on the LR(k) machine.

Graphs A directed graph consists of a set of vertices and a set of edges. An edge directed from p to q with a label X is denoted by $p \xrightarrow{X} q$. A path $p \overset{\alpha}{\rightsquigarrow} q$ is a sequence of vertices and edges leading from p to q with a label sequence α. Each label X has the associated cost. The length of path $p \overset{\alpha}{\rightsquigarrow} q$ is the sum of the label costs on its edges. If vertices p and q are connected, the distance from p to q is the minimum length among all the possible paths from p to q. If p and q are not connected, the distance is defined to be infinite.

Uniform cost search Uniform cost search algorithm [19,22] is a varied best-first graph search algorithm. This algorithm evaluates nodes using $f(n)$, which evaluates the cost of getting from the start node to node n. This algorithm chooses the next node which has the least $f(n)$. This process repeats until a goal node is reached along some path. As the least-cost path was always the one chosen for extension, the path first reaching the goal is sure to be the optimal path. Figure 3 shows the uniform cost search algorithm.

Let u_s be the start node.
Let Q be $\{(u_s, 0)\}$.
while Q is not empty **do**
  remove $(u_1 \overset{\alpha}{\rightsquigarrow} u_i, c_f)$ whose c_f is minimal from Q
  if u_i is a goal **then**
    break the loop
  foreach $u_i \xrightarrow{X} u_j$ **do**
    insert $(u_1 \overset{\alpha}{\rightsquigarrow} u_i \xrightarrow{X} u_j, c_f + \text{cost}(X))$ into Q

Fig. 3 Uniform cost search algorithm: $\text{cost}(X) \geq 0$

Let u_s be the start node.
Let Q be $\{(u_s, 0, g(u_s))\}$.
while Q is not empty **do**
  remove $(u_1 \overset{\alpha}{\rightsquigarrow} u_i, c_f, c_g)$ whose $c_f + c_g$ is minimal from Q
  if u_i is a goal **then**
    break the loop
  foreach $u_i \xrightarrow{X} u_j$ **do**
    insert $(u_1 \overset{\alpha}{\rightsquigarrow} u_i \xrightarrow{X} u_j, c_f + \text{cost}(X), g(u_j))$ into Q

Fig. 4 The A* algorithm: $\text{cost}(X) \geq 0$

The A algorithm* The A* algorithm [11,12,17,19,22] is a best-first graph search algorithm that improves the uniform cost search algorithm greatly. This algorithm evaluates nodes using $f(n) + g(n)$ where

- $f(n)$ is the cost of reaching node n from the start node, and
- $g(n)$ is an estimated cost of getting from node n to a goal node.

The $f(n) + g(n)$ is the estimated cost of the least-cost path to a goal node through n. This algorithm chooses the next node n whose $f(n) + g(n)$ is minimal. This process repeats until a goal node is reached. Figure 4 shows a simplified A* algorithm.

This algorithm always finds a least-cost path from the start node to a goal node when such a path exists if $g(n)$ never overestimates the cost of getting to a goal node [11,12,17,19,22]. That is, if the real minimum cost of getting from node n to a goal node is $C(n)$, then $g(n)$ satisfies $g(n) \leq C(n)$.

Generally speaking, depth-first search, breadth-first search, and uniform cost search [19] are special cases of the A* algorithm. For example, if $g(n) = 0$ for all nodes, this algorithm will behave on a general graph like the uniform cost search algorithm.

3 Efficient error repair method using the A* algorithm

Our repair method applies insertions, deletions, or unrestricted shifts to a portion of remaining input string until parsing can either continue farthest or leads to an accepting configuration. Furthermore, our repair method uses the A* algorithm, allows nonterminal insertions, and removes unproductive or redundant configurations in order to speed up repair and to reduce memory use. Our method improves on McKenzie et al.'s method (p. 15) Cerecke's method (p. 16), and Corchuelo et al.'s method (p. 6, 17).

In the following, we will improve Corchuelo et al.'s method [5] by removing unproductive configurations, allowing inserting nonterminals, applying the A* algorithm, removing redundant configurations, and improving the repair quality.

3.1 Corchuelo et al.'s method

This method attempts to apply a sequence of insertions, deletions or restricted shifts to a portion of the remaining input string until parsing can continue for some fixed number of symbols or else parsing leads to an accepting configuration. This method improves the repair quality of McKenzie et al.'s method and Cerecke's method by allowing restricted shifts.

This method uses three transition rules in Fig. 5 to generate new configurations. Configuration $(q_1 \ldots q_i, p, e, c)$ is composed of a state sequence $(q_1 \ldots q_i)$, the current position of

$$\texttt{(CO-INS)} \quad \frac{(q_1 \ldots q_i,\ aa_p \ldots a_m) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j,\ a_p \ldots a_m)}{(q_1 \ldots q_i\ , p,\ e,\ c) \xrightarrow{\text{CO}} (q_1 \ldots q_j,\ p,\ e \cdot (\text{ins}\ a),\ c + ic(a))}$$

$$\texttt{(CO-DEL)} \quad \frac{a_p \neq \$}{(q_1 \ldots q_i,\ p,\ e,\ c) \xrightarrow{\text{CO}} (q_1 \ldots q_i,\ p+1,\ e \cdot (\text{del}\ a_p),\ c + dc(a_p))}$$

$$\texttt{(CO-SHIFT)} \quad \frac{\begin{array}{c}(q_1 \ldots q_i,\ a_p \ldots a_m) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j,\ a_{p+k} \ldots a_m) \\ k = N \ \vee\ (0 < k < N \wedge \text{action}[q_j, a_{p+k}] \in \{\text{accept}, \text{error}\})\end{array}}{(q_1 \ldots q_i,\ p,\ e,\ c) \xrightarrow{\text{CO}} (q_1 \ldots q_j,\ p+k,\ e \cdot \underbrace{(\text{shift}) \cdots (\text{shift})}_{k},\ c))}$$

Fig. 5 Corchuelo et al.'s transition rules

the first remaining input symbol (p), the sequence of edit operations have been made so far (e), and the sum of insertion and deletion costs to reach the current configuration (c).

This error repair method is based on the uniform cost search algorithm. The priority queue has the initial configuration $(q_1 \ldots q_i, p, \varepsilon, 0)$ for a given error configuration $(q_1 \ldots q_i, a_p \ldots a_m)$. This method removes the least-cost configuration from the priority queue, generates new configurations from the least-cost one by using transition rules in Fig. 5, and inserts the new generated ones into the priority queue. This process repeats until parsing can either continue for at least N symbols or leads to an accepting configuration.

Meanwhile, `(CO-SHIFT)` allows only the restricted shift operations in that this rule generates only the configuration obtained after applying sequential k-shift operations but does not generate all intermediate configurations obtained by m-shift operations ($0 < m < k$). This restricted shift operation sometimes causes the parser to miss the proper repair candidates. Supporting unrestricted shift operations seems trivial by generating all intermediate configurations, but it can degrade the repair performance severely because of increased generated configurations.

3.2 Removing unproductive configurations

We will remove unproductive configurations that Corchuelo et al.'s method [5], Cerecke's method [4], and McKenzie et al.'s method [16] generate. Configurations are unproductive if they never lead to a goal configuration. Removing unproductive configurations can not only speed up the repair process but also reduce required memory space.

Insertion rules can generate unproductive configurations in Corchuelo et al.'s method [5] Cerecke's method [4] and McKenzie et al.'s method [16]. To formally handle generated insertion strings, we introduce two sets of terminal strings. Follow$(q_1 \ldots q_i)$ is the set of terminal strings that follow stack sequence $q_1 \ldots q_i$, and Follow$_a(q_1 \ldots q_i)$ is the set of prefixes of Follow$(q_1 \ldots q_i)$ that end with the terminal symbol a.

Definition 1

$$\begin{aligned}&\text{Follow}(q_1 \ldots q_i) \\ &\quad = \{x \in T^* \mid (q_1 \ldots q_i, x\$) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j, \$), \text{action}[q_j, \$] = \text{accept}\} \\ &\text{Follow}_a(q_1 \ldots q_i) \\ &\quad = \{xa \in T^* \mid (q_1 \ldots q_i, xay\$) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j, y\$)\}\end{aligned}$$

$$\texttt{(R1I)} \quad \frac{axa_p \in \text{Follow}_{a_p}(q_1 \dots q_i)}{(q_1 \dots q_i,\ p,\ e,\ c) \xrightarrow{\text{R1}} (q_1 \dots q_j,\ p,\ e \cdot (\text{ins } a),\ c + ic(a))}$$

$$\texttt{(R1D)} \quad \frac{a_p \neq \$}{(q_1 \dots q_i,\ p,\ e,\ c) \xrightarrow{\text{R1}} (q_1 \dots q_i,\ p+1,\ e \cdot (\text{del } a_p),\ c + dc(a_p))}$$

$$\texttt{(R1S)} \quad \frac{\begin{array}{c}(q_1 \dots q_i,\ a_p \dots a_m) \xrightarrow{\text{LR}}{}^{*} (q_1 \dots q_j,\ a_{p+k} \dots a_m) \\ k = N \ \vee \ (0 < k < N \wedge \text{action}[q_j, a_{p+k}] \in \{\text{accept}, \text{error}\})\end{array}}{(q_1 \dots q_i,\ p,\ e,\ c) \xrightarrow{\text{R1}} (q_1 \dots q_j,\ p+k,\ e \cdot \underbrace{(\text{shift}) \cdots (\text{shift})}_{k},\ c)}$$

Fig. 6 ⟪R1⟫: Transition rules to prune unproductive configurations

Using Definition 1, insertion rule (CO-INS) in Corchuelo et al.'s method (or insertion rule (MC-INS) in McKenzie et al.'s method on p. 15) is expressible as

$$\frac{ax \in \text{Follow}(q_1 \dots q_i)}{(q_1 \dots q_i,\ p,\ e,\ c) \xrightarrow{\text{CO}} (q_1 \dots q_j,\ p,\ e \cdot (\text{ins } a),\ c + ic(a))}.$$

However, there can be some terminal a where $ax \in \text{Follow}(q_1 \dots q_i)$ for some terminal string x but $aya_p \notin \text{Follow}_{a_p}(q_1 \dots q_i)$ for any terminal string y. Inserting such terminal symbol a brings about an unproductive configuration that cannot lead to a goal configuration $(q_1 \dots q_j, p, e, c)$ where action$[q_j, a_p] \in \{\texttt{shift}\ q, \texttt{accept}\}$, as in McKenzie et al.'s method [16], Cerecke's method [4], and Corchuelo et al.'s method [5].

We can avoid unproductive configurations in repairing syntax errors. Figure 6 shows the new rule set ⟪R1⟫ for repairs. To eliminate unproductive configurations, we restrict insertion symbols. In other words, we insert only terminal a satisfying $axa_p \in \text{Follow}_{a_p}(q_1 \dots q_i)$ instead of $ax \in \text{Follow}(q_1 \dots q_i)$ as in rule (R1I).

$$\frac{axa_p \in \text{Follow}_{a_p}(q_1 \dots q_i)}{(q_1 \dots q_i,\ p,\ e,\ c) \xrightarrow{\text{R1}} (q_1 \dots q_j,\ p,\ e \cdot (\text{ins } a),\ c + ic(a))}$$

Excluding rule (R1I), rule set ⟪R1⟫ is similar to Corchuelo et al.'s transition rules. Rule set ⟪R1⟫ generates configuration $(q_1 \dots q_i, p, e, c)$ that comprises a state sequence $(q_1 \dots q_i)$, the current position in the remaining input symbols (p), the sequence of edit operations have been made so far (e), and the sum of insertion and deletion costs to reach the current configuration (c).

As we will continue improving on ⟪R1⟫ in the subsequent subsections, the specific algorithm to compute $\text{Follow}_{a_p}(q_1 \dots q_i)$ is not presented here. In Sect. 3.4, we will remove unproductive configurations using the A* algorithm.

3.3 Inserting nonterminal symbols

By inserting nonterminal symbols as well as terminal symbols, we will remove reduce actions causing many redundant configurations that Corchuelo et al. method [5] and McKenzie et al. method [16] generate. Removing redundant configurations speeds up the repair process and reduces the required memory space.

Configurations are redundant if their costs are not the least-cost one among their equivalent configurations. Configurations are equivalent if they have the identical state sequence and remaining input string. We can always apply the same insertion, deletion, and shift

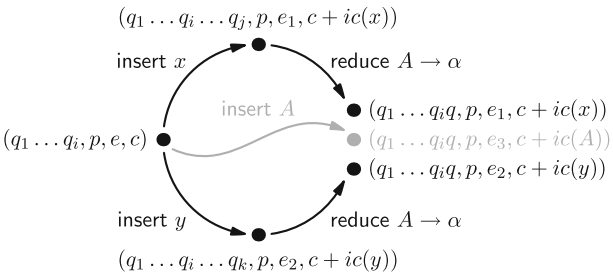


Fig. 7 Suppose $[B \to \beta_1 \bullet A\beta_2] \in q_i$, $[A \to \bullet\alpha] \in q_i$, action$[q_i\ A] = \texttt{goto}\ q$, $\alpha \Rightarrow^* x$ and $\alpha \Rightarrow^* y$ for the current configuration $(q_1 \ldots q_i,\ p, e, c)$. Inserting terminal strings x and y requires to reduce $A \to \alpha$, which causes equivalent configurations. If inserting A instead of reducing $A \to \alpha$, we obtain the least-cost configuration $(q_1 \ldots q_i q,\ p, e_3, c + ic(A))$ removing redundant configurations $(q_1 \ldots q_i q,\ p, e_1, c + ic(x))$ and $(q_1 \ldots q_i q,\ p, e_2, c + ic(y))$

operations to equivalent configurations, and after applying the same operations we shall obtain new equivalent configurations. Removing equivalent configurations except the least-cost one (or removing redundant configurations) is desirable for reducing search space and speeding up the repair process.

Inserting terminal symbols alone entails reduce actions that bring about redundant configurations. Suppose that the state sequence of the current configuration is $q_1 \ldots q_i$ and two items $[B \to \alpha \bullet A\beta]$ and $[A \to \bullet\alpha]$ are in state q_i. Then, reducing $A \to \alpha$ after inserting terminal strings x and y that α generates can make the current configuration lead to equivalent configurations with the same state sequence of $q_1 \ldots q_i q$ where action$[q_i, A] = \texttt{goto}\ q$ and the same current position p in the remaining input string as in Fig. 7.

Inserting nonterminal symbols, as well as terminal symbols, can remove reduce actions causing redundant configurations. Suppose that we insert A at the current configuration instead of reducing $A \to \alpha$ at $(q_1 \ldots q_i \ldots q_j,\ p, e_1, c + ic(x))$ and $(q_1 \ldots q_i \ldots q_k,\ p, e_2, c + ic(y))$ in Fig. 7. Then, we can avoid generating redundant configurations $(q_1 \ldots q_i q,\ p, e_1, c + ic(x))$ and $(q_1 \ldots q_i q,\ p, e_2, c + ic(y))$, and obtain the new least-cost configuration $(q_1 \ldots q_i q,\ p, e_3, c + ic(A))$ instead. (Note that $ic(A) \leq ic(x)$ for any terminal string x generated by A from the definition of nonterminal's insertion cost.)

In computing Follow$_a(q_1 \ldots q_i)$, we will remove reduce actions by using nonterminal symbols as well as terminal symbols.

Theorem 1

$$
\begin{aligned}
&\text{Follow}_a(q_1 \ldots q_i) \\
&= \{xa \in T^* \mid q_i \overset{\alpha}{\rightsquigarrow} q \xrightarrow{a} q' \ \ \textit{in LR machine},\ \alpha a \Rightarrow^* xa\} \\
&\cup \bigcup_{[A \to \alpha\bullet\beta] \in \text{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}_a(q_1 \ldots q_{i-|\alpha|} q) \\
&\qquad \textit{where}\ \text{action}[q_{i-|\alpha|}, A] = \texttt{goto}\ q
\end{aligned}
$$

Proof The proof is in the "Appendix".

Theorem 1 shows that Follow$_a(q_1 \ldots q_i)$ is computable by using terminal and nonterminal symbols on the shift and goto paths in the LR-machine. Furthermore, in computing Follow$_a(q_1 \ldots q_i)$, Theorem 1 uses only kernel items for reduce actions, and removes all the reduce actions caused by non-kernel items.

Using Theorem 1, we can remove reduce actions by inserting nonterminal symbols as well as terminal symbols in repairing syntax errors. Figure 8 shows the new transition rule

$$\texttt{(R2I-S)}\quad \frac{q_i \overset{X\alpha}{\rightsquigarrow} q \xrightarrow{a_p} q'}{(q_1 \ldots q_i\ , p,\ t,\ e,\ c) \xrightarrow{\text{R2}} (q_1 \ldots q_i q,\ p,\ t+1,\ e \cdot (\text{ins } X),\ c + ic(X))}$$

$$\texttt{(R2I-R)}\quad \frac{[A \rightarrow \alpha \bullet \beta] \in kernel(q_i) \qquad \text{action}[q_{i-|\alpha|}, A] = \texttt{goto}\ q}{(q_1 \ldots q_i,\ p,\ 1,\ e,\ c) \xrightarrow{\text{R2}} (q_1 \ldots q_{i-|\alpha|} q,\ p,\ 1,\ e \cdot (\text{ins } \beta),\ c + ic(\beta))}$$

$$\texttt{(R2D)}\quad \frac{a_p \neq \$}{(q_1 \ldots q_i,\ p,\ 1,\ e,\ c) \xrightarrow{\text{R2}} (q_1 \ldots q_i,\ p+1,\ 1,\ e \cdot (\text{del } a_p),\ c + dc(a_p))}$$

$$\texttt{(R2S)}\quad \frac{\begin{array}{c}(q_1 \ldots q_i,\ a_p \ldots a_m) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j,\ a_{p+k} \ldots a_m) \\ k = N \ \vee\ (0 < k < N \wedge \text{action}[q_j, a_{p+k}] \in \{\text{accept}, \text{error}\})\end{array}}{(q_1 \ldots q_i,\ p,\ t,\ e,\ c) \xrightarrow{\text{R2}} (q_1 \ldots q_j,\ p+k,\ 1,\ e \cdot \underbrace{(\text{shift}) \cdots (\text{shift})}_{k},\ c)}$$

Fig. 8 ⟪R2⟫: Transition rules to allow nonterminal transitions

set ⟪R2⟫ to allow nonterminal insertions as well. Transition rule set ⟪R2⟫ generates configuration $(q_1 \ldots q_i, p, t, e, c)$ that comprises a state sequence $(q_1 \ldots q_i)$, the current position of the first remaining input symbol (p), the number of successive shift or goto transitions that the current configuration has had since the last reduction (t), the sequence of edit operations have been made so far (e), and the sum of insertion and deletion costs to reach the current configuration (c). Rule `(R1I)` in ⟪R1⟫ changes into `(R2I-S)` and `(R2I-R)` in ⟪R2⟫. Rule `(R2I-S)` inserts a vocabulary symbol X for repairs, and rule `(R2I-R)` brings about reduce actions using the kernel items only. Rule set ⟪R2⟫ removes reduce actions non-kernel items cause that generate many redundant configurations.

3.4 Applying the A* algorithm

To accelerate the repair process and to prune unproductive configurations, we will adopt the A* algorithm instead of the uniform cost search algorithm. The A* algorithm makes repair process faster than the uniform cost search algorithm, and is also helpful to remove the inefficiency that rule `(R2I-S)` has to check if there is a path satisfying $q_i \overset{X\alpha}{\rightsquigarrow} q \xrightarrow{a_p} q'$ for every visiting configuration $(q_1 \ldots q_i, p, t, e, c)$ before inserting a vocabulary symbol X. Furthermore, the A* algorithm can remove unproductive configurations.

Heuristic functions are indispensable to the A* algorithm. Heuristic functions guess the distance to the least-cost goal configuration. As the guessed distance gets closer to the real distance, we can achieve better performance. However, the guessed distance must not overestimate the real distance in order to always obtain the least-cost repair. Heuristic functions can also prune unproductive configurations. Pruning unproductive configurations decreases memory use and accelerates the repair process because it greatly diminishes the number of generated configurations.

We define the guessed distance using the property that reduce actions or deletions do not occur if $t > 1$ for configuration $(q_1 ... q_i, p, t, e, c)$ in ⟪R2⟫. Rule `(R2I-S)` shows that configuration $(q_1 ... q_i, p, t, e, c)$ uses only shift or goto actions to reach the goals if $t > 1$. In this case, we know the real distance to the least-cost goal as follows.

Definition 2

$$\begin{aligned}&\text{dist}(q, a)\\ &= \min\{ic(\gamma) \mid q \overset{\gamma\alpha}{\rightsquigarrow} q' \text{ in LR machine}\} && \text{if some } q \overset{\gamma a}{\rightsquigarrow} q' \text{ exists in LR machine}\\ &= \infty && \text{otherwise}\end{aligned}$$

$$\texttt{(R3I-S)}\ \frac{q_i \xrightarrow{X} q \qquad \mathrm{dist}(q, a_p) < \infty}{(q_1 \ldots q_i, p, t, e, c_f, c_g) \xrightarrow{\mathrm{R3}} (q_1 \ldots q_i q, p, t+1, e \cdot (\mathrm{ins}\ X), c_f + ic(X), \mathrm{dist}(q, a_p))}$$

$$\texttt{(R3I-R)}\ \frac{[A \to \alpha \bullet \beta] \in kernel(q_i) \qquad \mathrm{action}[q_{i-|\alpha|} A] = \texttt{goto}\ q}{(q_1 \ldots q_i,\ p,\ 1,\ e,\ c_f,\ c_g) \xrightarrow{\mathrm{R3}} (q_1 \ldots q_{i-|\alpha|} q,\ p,\ 1,\ e \cdot (\mathrm{ins}\ \beta),\ c_f + ic(\beta),\ 0)}$$

$$\texttt{(R3D)}\ \frac{a_p \neq \$}{(q_1 \ldots q_i,\ p,\ 1,\ e,\ c_f,\ c_g) \xrightarrow{\mathrm{R3}} (q_1 \ldots q_i,\ p+1,\ 1,\ e \cdot (\mathrm{del}\ a_p),\ c_f + dc(a_p),\ 0)}$$

$$\texttt{(R3S)}\ \frac{\begin{array}{c}(q_1 \ldots q_i,\ a_p \ldots a_m) \xrightarrow{\mathrm{LR}^*} (q_1 \ldots q_j,\ a_{p+k} \ldots a_m) \\ k = N \ \vee\ (0 < k < N \wedge \mathrm{action}[q_j, a_{p+k}] \in \{\mathrm{accept}, \mathrm{error}\})\end{array}}{(q_1 \ldots q_i,\ p,\ t,\ e,\ c_f,\ c_g) \xrightarrow{\mathrm{R3}} (q_1 \ldots q_j,\ p+k,\ 1,\ e \cdot \underbrace{(\mathrm{shift}) \cdots (\mathrm{shift})}_{k},\ c_f,\ 0)}$$

Fig. 9 $\langle\!\langle \texttt{R3} \rangle\!\rangle$: Transition rules to use the A* algorithm

Hence, our guessed distance using $\mathrm{dist}(q, a)$ is as follows. Our guessed distance is $\mathrm{dist}(q_i, a_p)$ if $t > 1$ for configuration $(q_1 ... q_i, p, t, e, c)$ because $\mathrm{dist}(q_i, a_p)$ is the real distance to the least-cost goal in this case. However, our guessed distance is 0 if $t = 1$ because reduce actions or deletions can possibly intervene for $(q_1 ... q_i, p, t, e, c)$ to reach the goals. As mentioned, the guessed distance must not overestimate the real distance to obtain the least-cost goal in the A* algorithm.

We can compute $\mathrm{dist}(q, a)$ into a table at parser generation time. When repairing syntax errors, we just look up the precalculated table for $\mathrm{dist}(q, a)$ instead of computing it every time. Looking up the precalculated table reduces the execution time of the repair process, while it requires additional storage. To store $\mathrm{dist}(q, a)$ into a table, we need a space of $|Q_M| \cdot |T \cup \{\$\}| \cdot B$ where Q_M is the set of all LR-states, T is the set of terminal symbols, and B is the bit size of the maximum value of $\mathrm{dist}(q, a)$.

Figure 9 shows the new transition rule set $\langle\!\langle \texttt{R3} \rangle\!\rangle$ that repairs syntax errors using the A* algorithm. Configuration $(q_1 ... q_i, p, t, e, c_f, c_g)$ contains the cost traversed already c_f, and the guessed distance to the least-cost goal c_g. Guessed distance c_g is 0 when $t = 1$. The total cost of the configuration is $c_f + c_g$. If $t > 1$ and $\mathrm{dist}(q_i, a_p)$ is infinite or there is no path satisfying $q_i \overset{\gamma a_p}{\rightsquigarrow} q'$ in LR-machine, then the configuration is unproductive since the configuration never reaches the goal configurations using shift or goto actions only. Such unproductive configurations can cause the huge space of configurations, and slow the error repair process. Rule `(R3I-S)` prunes unproductive configurations by not generating new configurations when $\mathrm{dist}(q, a_p)$ is infinite. Meanwhile, rule `(R3I-S)` is more efficient than rule `(R2I-S)`. That is because rule `(R3I-S)` checks only whether $\mathrm{dist}(q, a_p) < \infty$ by looking up the precomputed table before generating new configurations, unlike rule `(R2I-S)` that checks whether there is a path satisfying $q_i \overset{X\alpha}{\rightsquigarrow} q \xrightarrow{a_p} q'$ for every visiting configuration $(q_1 \ldots q_i, p, t, e, c)$.

3.5 Removing redundant configurations

We will remove redundant configurations that transition rules in $\langle\!\langle \texttt{R3} \rangle\!\rangle$ still generate. Although we eliminated redundant configurations that non-kernel items cause in Sect. 3.3, there can still be some redundant configurations that kernel items bring about. We will remove some of such redundant configurations not only to speed up the repair process but also to reduce required memory space.

We use the hash table to remove the redundant configurations. For configuration $(q_1 \ldots q_i, p, t, e, c_f, c_g)$, we use $(q_1 \ldots q_i, p)$ as an index to the hash table. We keep

only the least-cost configurations in the hash table by removing other redundant ones. If $(q_1 \ldots q_i, p, t', e', c'_f, c'_g)$ is going to be inserted when $(q_1 \ldots q_i, p, t, e, c_f, c_g)$ is already in the hash table, then the configuration with a greater cost will be removed and the configuration with a less cost will be stored at the hash table because $(q_1 \ldots q_i, p, t, e, c_f, c_g)$ and $(q_1 \ldots q_i, p, t', e', c'_f, c'_g)$ are equivalent.

We remove only the redundant configurations that rule `(R3I-R)` generates. For every generated configuration, looking up its equivalent configuration in the hash table is expensive because the computation requires comparing the state sequence of the generated configuration with the state sequences of a group of configurations having the same hash value in the hash table. Hence, if we have a redundancy test for every generated configuration, the repair process will get slow. On the other hand, if we have no redundancy test for any generated configurations, the repair process will get slow too. That is because redundant configurations will increase greatly. For these reasons, we have a redundancy test for only the configurations that rule `(R3I-R)` generates.

3.6 Improving the repair quality

We alter rule `(R3S)` in $\langle\!\langle \texttt{R3} \rangle\!\rangle$ to

$$(\texttt{R3S}-\texttt{n})\ \frac{(q_1 \ldots q_i,\ a_p \ldots a_m) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j,\ a_{p+1} \ldots a_m)}{(q_1 \ldots q_i,\ p,\ t,\ e,\ c_f,\ c_g) \xrightarrow{\text{R3}} (q_1 \ldots q_j,\ p+1,\ 1,\ e \cdot (\text{shift}),\ c_f,\ 0)}$$

in order to improve the repair quality. Rule `(R3S-n)` allows examining more varied configurations than rule `(R3S)` by generating every configuration after each possible shift operation. Unlike `(R3S-n)`, rule `(R3S)` generates only the configuration after k shifts; it does not generate intermediate configurations after m shifts for $0 < m < k$. As a result, rule `(R3S)` degrades the repair quality because it does not generate intermediate configurations that would lead to more proper repairs.

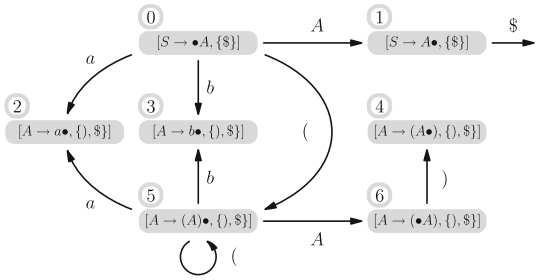
We also change the selection method to choose the repair among candidates. We consider a repair is a sequence of edit operations applying rules `(R3I-S)`, `(R3I-R)` and `(R3D)` within the first N_t input symbols and applying rule `(R3S-n)` to any possible locations such that after applying the repair, parsing can continue farther than after applying other repair candidates. Once a repair candidate is selected, other repair candidates with greater costs will be removed. Unlike our selection method, Corchuelo et al.'s method considers a repair is a sequence of edit operations using rules `(CO-INS)`, `(CO-DEL)` and `(CO-SHIFT)` within the first N_t input symbols such that after applying the repair, parsing can continue for at least N symbols or else it reaches an accepting configuration.

3.7 Examples

We will illustrate how our method works when it detects a syntax error, and compare our method with Cerecke's method (p. 16) and Corchuelo et al.'s method (p. 6, 17). Let $G = (N, T, P, S)$ be a context-free grammar such that

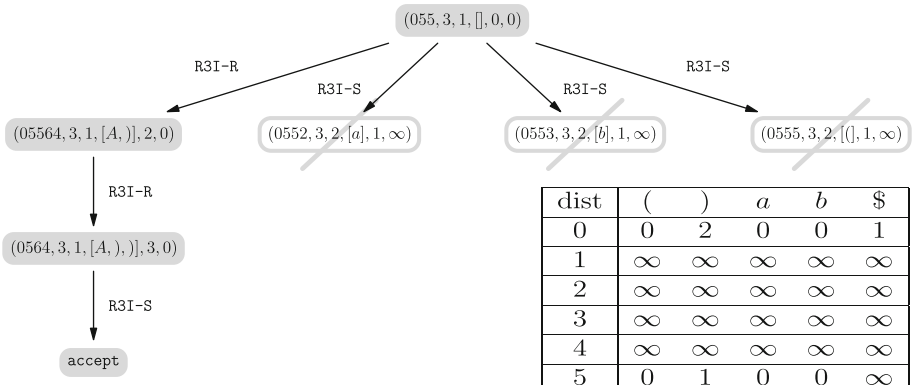
- $N = \{S, A\}$,
- $T = \{a, b, (,)\}$, and
- $P = \{S \to A, A \to (A), A \to a, A \to b\}$.

Figure 10 shows the LR(1)-machine and parsing table for G. LR states show their corresponding kernel items. Let $x = ((\$$ be the input string. Then, the LR(1) parser will detect a



	(	)	a	b	$	A
0	sht 5		sht 2	sht 3		goto 1
1					accept	
2		rd $A \to a$			rd $A \to a$	
3		rd $A \to b$			rd $A \to b$	
4		rd $A \to (A)$			rd $A \to (A)$	
5	sht 5		sht 2	sht 3		goto 6
6		sht 4				

Fig. 10 The LR(1)-machine and parsing table for G. LR states show their corresponding kernel items



dist	(	)	a	b	$
0	0	2	0	0	1
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	0	1	0	0	∞
6	∞	0	∞	∞	∞

Fig. 11 Configuration tree using ⟪R3⟫ and the table for dist(q, a) for input string (($. Configuration $(q_1 \ldots q_i, p, t, e, c_f, c_g)$ is composed of a state sequence $(q_1 ... q_i)$, the current position of the first remaining input symbol (p), the number of last states whose shift or goto actions have been considered (t), the sequence of edit operations have been made so far (e), the cost traversed already c_f, and the guessed distance to the least-cost goal c_g. The unproductive configurations (0552, 3, 2, [a], 1, ∞), (0553, 3, 2, [b], 1, ∞) and (0555, 3, 2, [(], 1, ∞) are not generated in fact

syntax error after parsing the first two symbols. That is,

$$(0, ((\$) \xrightarrow{\text{LR}} (05, (\$) \xrightarrow{\text{LR}} (055, \$) \longrightarrow \texttt{error}.$$

Our method using ⟪R3⟫ starts from configuration (055, 3, 1, [ ], 0, 0) and generates other configurations using `(R3I-S)` and `(R3I-R)` in this example until it reaches a goal configuration. (In this case, it reaches the accepting configuration.) All generated configurations are shown in Fig. 11. The unproductive configurations (0552, 3, 2, [a], 1, ∞), (0553, 3, 2, [b], 1, ∞) and (0555, 3, 2, [(], 1, ∞) are not generated in fact because their c_g values are infinite. Our method generates only four configurations, which is more efficient than Corchuelo et al.'s method in Fig. 12 and Cerecke's method in Fig. 13.

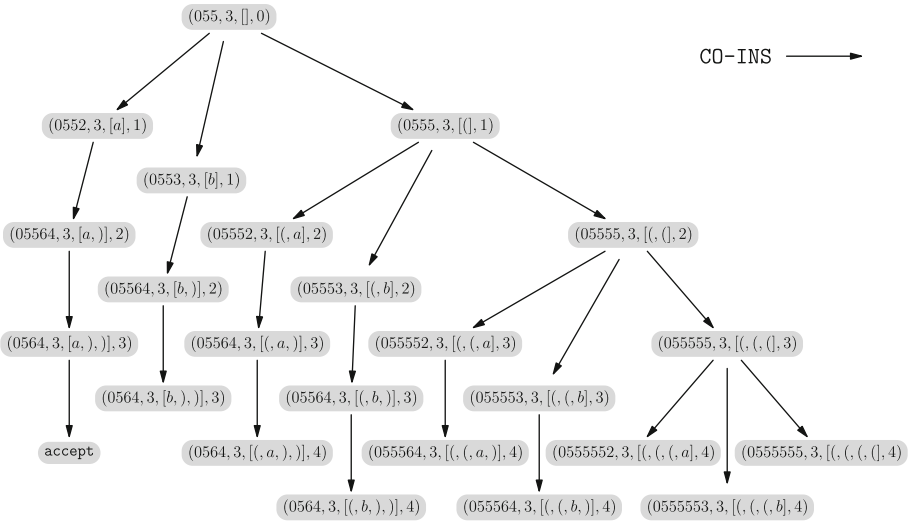


Fig. 12 Configuration tree using Corchuelo et al.'s method for input string (($. Configuration ($q_1 \ldots q_i$, p, e, c) is composed of a state sequence ($q_1 ... q_i$), the current position of the first remaining input symbol (p), the sequence of edit operations have been made so far (e) and the sum of insertion and deletion costs to reach the current configuration (c)

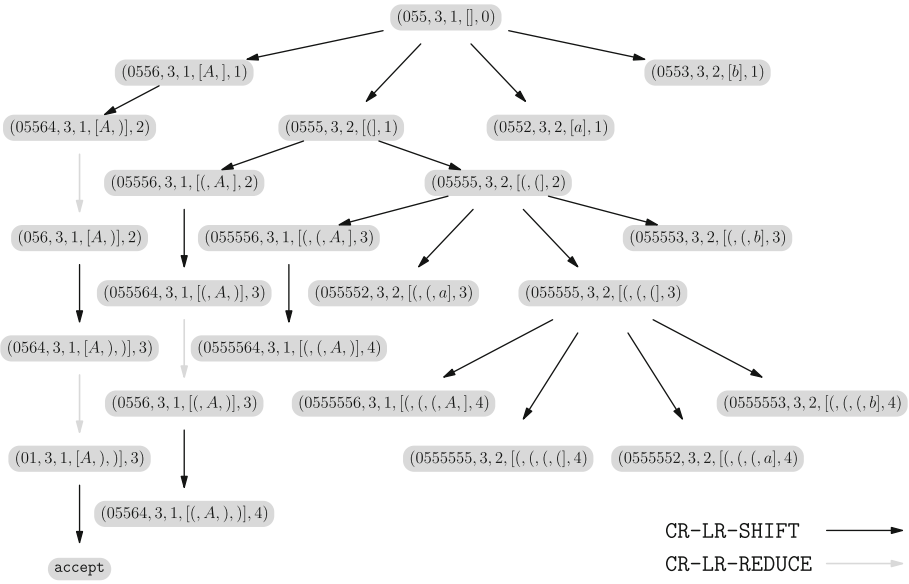


Fig. 13 Configuration tree using Cerecke's method for input string (($. Configuration ($q_1 \ldots q_i$, p, t, e, c) is composed of a state sequence ($q_1 ... q_i$), the current position of the first remaining input symbol (p), the number of last states whose shift or goto actions have been considered (t), the sequence of edit operations have been made so far (e) and the sum of insertion and deletion costs to reach the current configuration (c)

Corchuelo et al.'s method and Cerecke's method generate more configurations than our method as in Figs. 12, 13. Corchuelo et al.'s method also starts from (055, 3, 1, [ ], 0, 0) and generates other configurations using `(CO-INS)` in this example until it reaches a goal

$$\texttt{(MC-INS)} \quad \frac{(q_1 \ldots q_i,\ aa_p \ldots a_m) \xrightarrow{\text{LR}}{}^{*} (q_1 \ldots q_j,\ a_p \ldots a_m)}{(q_1 \ldots q_i\ , p,\ e,\ c) \xrightarrow{\text{MC}} (q_1 \ldots q_j,\ p,\ e \cdot (\text{ins}\ a),\ c + ic(a))}$$

$$\texttt{(MC-DEL)} \quad \frac{a_p \neq \$}{(q_1 \ldots q_i,\ p,\ e,\ c) \xrightarrow{\text{MC}} (q_1 \ldots q_i,\ p+1,\ e \cdot (\text{del}\ a_p),\ c + dc(a_p))}$$

Fig. 14 McKenzie et al.'s transition rules

configuration. Figure 12 shows the generated configurations using Corchuelo et al.'s method. This method generates 24 configurations in the worst case. Similarly, Cerecke's method also starts from (055, 3, 1, [ ], 0, 0) and generates other configurations using `(CR-LR-SHIFT)` and `(CR-LR-REDUCE)` here until it reaches a goal configurations. Figure 13 shows the generated 24 configurations in the worst case using Corchuelo et al.'s method.

4 Related works

Fischer et al.'s table-driven local error repair methods [7–10,15] attempt to edit an erroneous input string to a syntactically correct one by inserting and deleting terminal symbols. Their methods find insertion terminal strings by using kernel items and precalculated tables. However, their methods do not take account of the validation process that selects the most adequate one among some number of least-cost repair candidates, and they suffer from redundant stack configurations. Fischer and Mauney use the hash table to remove redundant stack configurations in LL(1) grammars [9].

Bertsch's method [2] removes redundant stack configurations without using the hash table, but the method does not consider terminal deletions and is applicable only to LL(1) grammars. Dain's method [6] generates the whole set of continuation strings of some fixed length first and then replaces a prefix of the remaining input string with the one whose distance is minimum according to the criterion by Wagner and Fischer [21]. However, the cost to find the whole set of continuation strings is very high. Penello and DeRemer [18] attempt to reduce redundant work by performing in parallel all possible parses from the position where a syntax error is found.

McKenzie et al.'s method [16] McKenzie et al.'s method attempts to edit an invalid input string to a syntactically correct one by inserting and deleting terminal symbols. In order to enhance the repair quality, this method takes account of the validation process that gathers some number of least-cost repair candidates and then chooses the most adequate one among them as the final repair.

McKenzie et al.'s method uses two transition rules in Fig. 14 where configuration $(q_1 \ldots q_i, p, e, c)$ is composed of a state sequence $(q_1 ... q_i)$, the current position of the first remaining input symbol (p), the sequence of edit operations have been made so far (e), and the sum of insertion and deletion costs to reach the current configuration (c).

McKenzie et al.'s method generates all the reachable configurations by using `(MC-INS)` and `(MC-DEL)` until some fixed number of least-cost goals are obtained. The goal is some configuration $(q_1 ... q_i, p, e, c)$ where action$[q_i, a_p] \in \{\texttt{shift}\ q, \texttt{accept}\}$ for a state q. This method will choose one among the obtained goals as the repair if the chosen one enables parsing to continue farthest.

McKenzie et al.'s method is easy to understand, and uses only the parsing table removing other precalculated ones (for instance, precalculated S and E tables used in [7–10].

$$\texttt{(CR-LR-SHIFT)}\quad \frac{q_i \xrightarrow{X} q}{(q_1 \ldots q_j,\ p,\ t,\ e,\ c) \xrightarrow{\text{CR}} (q_1 \ldots q_j,\ p,\ t+1,\ e \cdot (\text{ins } X),\ c + ic(X))}$$

$$\texttt{(CR-LR-REDUCE)}\quad \frac{[A \to \alpha\bullet] \in q_i \qquad |\alpha| \geq t \qquad \text{action}[q_{i-|\alpha|}, A] = \texttt{goto } q}{(q_1 \ldots q_i,\ p,\ t,\ e,\ c) \xrightarrow{\text{CR}} (q_1 \ldots q_{i-|\alpha|} q,\ p,\ 1,\ e,\ c)}$$

$$\texttt{(CR-DEL)}\quad \frac{a_p \neq \$}{(q_1 \ldots q_i,\ p,\ t,\ e,\ c) \xrightarrow{\text{CR}} (q_1 \ldots q_i,\ p+1,\ t,\ e \cdot (\text{del } a_p),\ c + dc(a_p))}$$

Fig. 15 Cerecke's transition rules

Theoretically this method can repair any syntax errors. However, this method generates redundant or unproductive configurations. Thus, for a huge number of generated configurations, this method often fails to find a repair within reasonable time and space. In case of failure, it reverts to a secondary recovery method such as panic mode [1], which degrades the repair quality severely. To discard redundant configurations, McKenzie et al. proposed the bitmap method [16] that removes configurations whose top states in their state sequences are the same except for the least-cost one. This bitmap method is very efficient but does not guarantee the least-cost error repair [3] and can not remove unproductive configurations.

Kim and Choe's method [14] Kim and Choe's method attempts to edit an erroneous input string to a syntactically correct one by insertion and deletion operations. This method takes account of inserting nonterminal symbols as well as terminal symbols for efficient error repairs. This method also considers the validation process in order to enhance the repair quality. This method is very efficient to repair syntax errors in $O(n + k \log n)$ where the k least-cost repair candidates are taken into account and the length of the parsing stack is n when a syntax error occurs.

Although Kim and Choe's method is very efficient, this method does not consider shift operations unlike Corchuelo et al.'s method [5]. In addition, though this method reduces redundant configurations greatly, it fails to remove all of them. As this method chooses the most proper one among the k least-cost repair candidates, the repair quality can be debased if there are some redundant configurations among the k least-cost repair candidates.

Cerecke's method [4] Cerecke's method improves McKenzie et al.'s method by inserting nonterminal symbols as well as terminal symbols. Inserting nonterminals can eliminate some reduce actions to bring about many redundant configurations occurring in McKenzie et al.'s method.

Cerecke's method uses three transition rules in Fig. 15 where configuration $(q_1 ... q_i,\ p, t, e, c)$ is composed of a state sequence $(q_1 ... q_i)$, the current position of the first remaining input symbol (p), the number of shift or goto actions this configuration has had since the last reduction (t), the sequence of edit operations have been made so far (e), and the sum of insertion and deletion to reach the current configuration (c).

Cerecke's method removes some reduce actions to cause redundant configurations. If $[A \to \alpha\bullet] \in q_i$ and $|\alpha| < t$ for configuration $(q_1 \ldots q_i, p, t, e, c)$, this method does not reduce $A \to \alpha$ at q_i because reducing it just generates a redundant configuration. Figure 16 depicts the situation. If $[A \to \alpha\bullet] \in q_i$ and $|\alpha| < t$ for configuration $(q_1 \ldots q_i, p, t, e, c)$, there must be some previous configuration $(q_1 ... q_{i-|\alpha|}, p, t - |\alpha|, e', c - ic(\alpha))$ leading to $(q_1 ... q_i, p, t, e, c)$ through insertion of α. Furthermore, from the previous configuration, configuration $(q_1 ... q_{i-|\alpha|} q, p, t - |\alpha| + 1, e'', c - ic(\alpha) + ic(A))$ was generated by insertion of A.

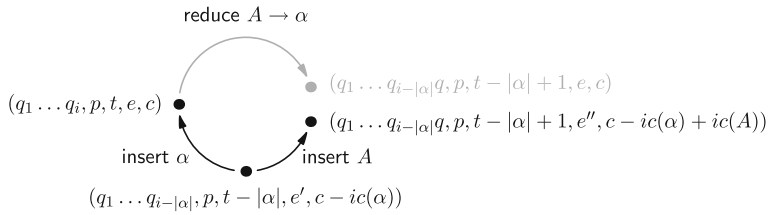


Fig. 16 Configuration $(q_1 \ldots q_{i-|\alpha|}q,\ p,\ t-|\alpha|+1,\ e,\ c)$ is redundant because it is equivalent to $(q_1 \ldots q_{i-|\alpha|}q,\ p,\ t-|\alpha|+1,\ e'',\ c-ic(\alpha)+ic(A))$ and its cost c is not less than $c-ic(\alpha)+ic(A)$

Reducing $A \rightarrow \alpha$ at q_i generates configuration $(q_1 \ldots q_{i-|\alpha|}q,\ p,\ t-|\alpha|+1,\ e,\ c)$, which is redundant because this configuration is equivalent to $(q_1 \ldots q_{i-|\alpha|}q,\ p,\ t-|\alpha|+1,\ e'',\ c-ic(\alpha)+ic(A))$ and its cost c is not less than $c-ic(\alpha)+ic(A)$. (Note that $ic(A) = \min\{ic(x) \mid A \Rightarrow^* x\}$ and $A \rightarrow \alpha$.) Hence, this unnecessary reduce action is avoidable.

Like McKenzie et al.'s method, Cerecke's method often fails to find a repair within reasonable time and space, so this method must have an additional way of reducing search space.

Corchuelo et al.'s method [5] This method is explained in Sect. 3.1. As mentioned, `(CO-SHIFT)` supports restricted shifts only. `(CO-SHIFT)` generates only the configuration obtained by applying sequential k-shift operations, but does not generate intermediate configurations obtained by m-shift operations ($m < k$). Such intermediate configurations can enhance the repair quality by considering more varied configurations whereas they slow the repair process by increasing the number of generated configurations. Furthermore, this method often fails to find a repair within reasonable time and space because this method generates unproductive and redundant configurations. In case of failure, it reverts to a secondary recovery mechanism such as panic mode [1], which degrades the repair quality severely.

5 Experimental results

This section shows the experimental results that compare our method with Cerecke's method [4], Kim and Choe's method [14] and Corchuelo et al.'s method [5] with respect to quality and efficiency. We exclude McKenzie et al.'s method from our experiments because Cerecke's method is similar to and improves on McKenzie et al.'s method by allowing nonterminal insertions. For brevity, we write `[NM]` for our new method, `[CE]` for Cerecke's method, `[KC]` for Kim and Choe's method, and `[CO]` for Corchuelo et al.'s method.

Our test programs are C and Java programs with syntax errors. For test programs, we corrupt syntactically correct 110 C programs and 144 Java programs by inserting, deleting or modifying symbols at random positions. Syntax errors are randomly made at one to five places in a program. Random syntax errors are made using the Unix's `rand()` function.

We carried out all the experiments on a machine running Linux kernel 2.6.20-1 with Intel Pentium-IV 3.20 GHz processor and 2 GB memory. For C language, we use the C grammar in the book *"The C Programming Language, Second Edition"* [13]. For Java language, we use the Java 1.1 grammar. For each erroneous input program, we iterate the same error repair 5 times, and average their total repair times.

When we estimate the repair efficiency using `[NM]`, we use rules `(R3I-S)`, `(R3I-R)`, `(R3D)` and `(R3S)` in $\langle\!\langle$`R3`$\rangle\!\rangle$ and adopt the Corchuelo et al.'s selection method of choosing

Table 1 Parameter settings for each experiment

Set	Method	Experiment	N_i	N_d	N_t	N	N_c
S1	[CE]	Mismatched parentheses	∞	3	–	3	500
	[KC]	(Figs. 17, 18, 19, 20)	∞	–	–	3	500
	[CO]		5	0	10	3	–
	[NM]		5	5	10	3	–
S2	[CE]	Repair efficiency	∞	3	–	3	500
	[KC]	(Figs. 21, 22, 23, 24, 25, 26)	∞	–	–	3	500
	[CO]		4	3	10	3	–
	[NM]		4	3	10	3	–
S3	[CE]	Repair quality	∞	3	–	30	500
	[KC]	(Table 2; Figs. 27, 28)	∞	–	–	30	500
	[CO]		4	3	10	3	–
	[NM]		100	100	30	–	–

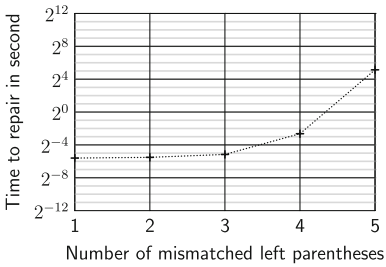


Fig. 17 Time to repair using [CE] in C

the repair among candidates for fair comparison with other methods. Meanwhile, when we estimate the repair quality using [NM], we use rules (R3I-S), (R3I-R), (R3D) and (R3S-n) and adopt the selection method in Sect. 3.6 for better repair quality.

We limit the repair process to a portion of the first N_t input symbols, the number of insertions to N_i, the number of deletions to N_d, the number of sequential shift operations to at most N and the number of distinct repair candidates to N_c. We use 1 for all insertion and deletion costs. Table 1 shows the parameter settings in each experiment. We set parameters differently in order to get the best results from each experiment within reasonable time and space.

To evaluate the repair efficiency, we measure the times to repair the following erroneous C programs

```
main () {int x; x=(((···(0;}
```

by increasing the number of mismatched left parentheses. This experiment shows our method, [NM], is much faster than [CE] and [CO] but slower than [KC] in repairing the above C programs. Figures 17, 18 show that [CE] and [CO] become slow exponentially as the number of mismatched left parentheses increases. (Note that the y-axes are log-scaled in Figs. 17, 18.) [CE] and [CO] become very slow if long insertion strings are needed to repair errors. Unlike these methods, Fig. 20 shows that [KC] is very fast to repair the above C programs. According to Fig. 20, [KC] becomes slow linearly as the number of mismatched

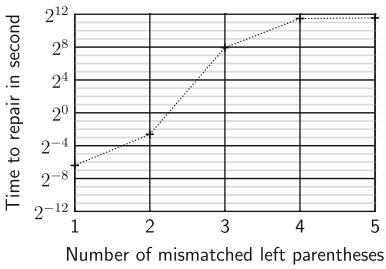


Fig. 18 Time to repair using `[CO]` in C

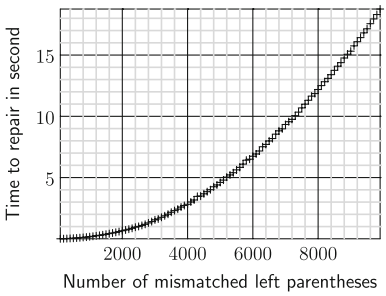


Fig. 19 Time to repair using `[NM]` in C

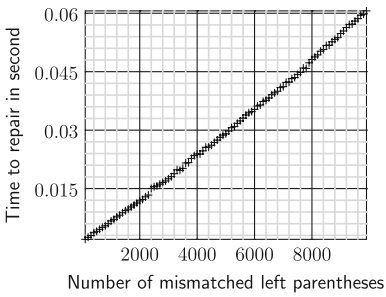


Fig. 20 Time to repair using `[KC]` in C

left parentheses increases. Figure 19 shows that `[NM]` still gets slow exponentially and is slower than `[KC]` but `[NM]` is much faster than `[CE]` and `[CO]`.

To evaluate the repair efficiency, we also measure the execution times to repair our test programs. Each point corresponds to a test program in Figs. 21, 22, 23, 24, 25, 26. The x and y coordinates of a point are the execution times to repair the corresponding test program using the x-axis method and the y-axis method, respectively. If a point (a, b) is located under the line $y = x$ in Figs. 21, 22, 23, 24, 25, 26, then $a > b$. Thus, the point's location implies that the y-axis method is more efficient to repair the corresponding test program than the x-axis method. As the x-axes are log-scaled in Figs. 21, 22, 23, 24, 25, 26 in order to show each point clearly, the line $y = x$ is shown as a curve in each graph.

Figures 21, 22, 23, 24, 25, 26 show that our method `[NM]` repairs syntax errors more rapidly than `[CE]`, `[KC]` and `[CO]` on the whole. Figures 21, 22, 23, 24, 25, 26 show X-marks to represent test programs that each method fails to repair. The X-marks at the rightmost of graphs imply that the x-axis method fails to repair. Likewise, the X-marks at the topmost of graphs imply that the y-axis method fails to repair. The `[CE]` method fails to repair one C program in Fig. 21 by consuming too much memory space. The `[NM]` method fails to repair

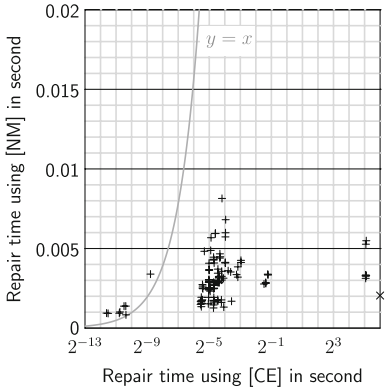


Fig. 21 Repair times for C programs

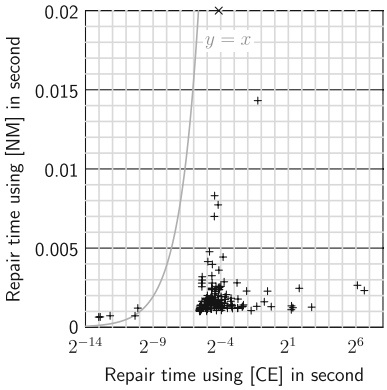


Fig. 22 Repair times for Java programs

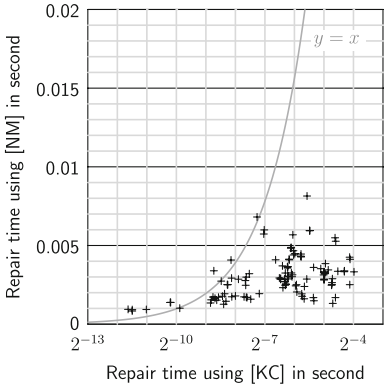


Fig. 23 Repair times for C programs

one Java program in Figs. 22, 24, 26 because of limited N_i and N_d. The `[CO]` method fails repair 4 C programs in Fig. 25 because of limited N_i and N_d.

In case of `[CO]`, we set $N_i = 4$, $N_d = 3$, $N_t = 10$ and $N = 3$ because Corchuelo et al. report that this setting produces good repairs [5]. One may think that `[CO]` might repair the 4 C programs if N_i and N_d become sufficiently large. In practice, however, `[CO]` still fails to find proper repairs within reasonable time and space because `[CO]` becomes very slow consuming too much memory space as $N_i > 4$ or $N_d > 3$.

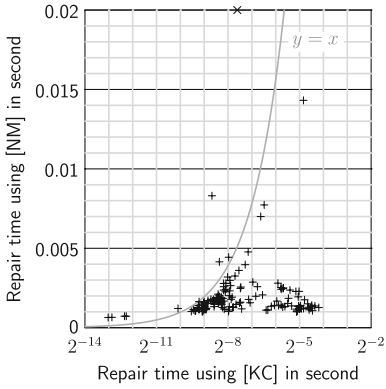


Fig. 24 Repair times for Java programs

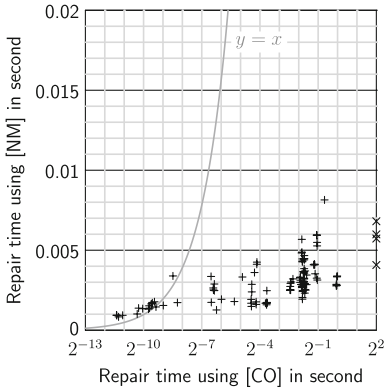


Fig. 25 Repair times for C programs

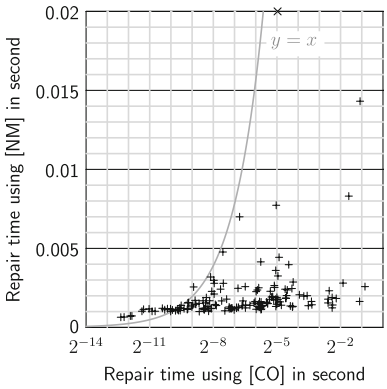


Fig. 26 Repair times for Java programs

Meanwhile, `[CO]` and `[NM]` using rule `(R3S)` should produce the same error repairs basically, but their numbers of failed test programs are different. That is because `[CO]` and `[NM]` select different repairs from the same cost of repair candidates. Carefully designing insertion and deletion costs can lessen the number of repair candidates with the same cost.

In order to evaluate the repair quality, we use the sum total of insertion and deletion costs to repair syntax errors in a test program. We assume that a method repairs a test program

Table 2 Repair costs and repair quality compared with [NM]

Method	Lang	Repair cost	Failure	< [NM] (%)	= [NM] (%)	> [NM] (%)
[CE]	C	549	–	0.9	74.5	24.5
[KC]	C	602	–	1.8	69.1	29.1
[CO]	C	539	3.6	1.8	68.2	30.0
[NM]	C	419	–	–	–	–
[CE]	Java	808	–	1.4	48.6	50.0
[KC]	Java	901	–	1.4	38.2	60.4
[CO]	Java	544	–	2.8	75.0	22.2
[NM]	Java	496	–	–	–	–

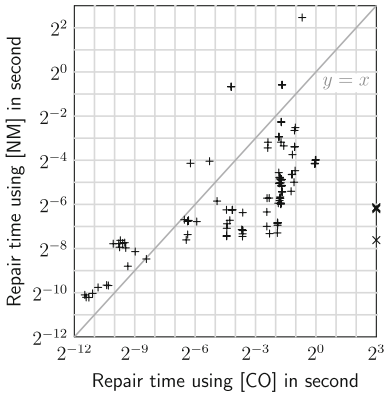


Fig. 27 Repair times for C programs

better than another if the method repairs the test program with less cost of edit operations than another. (Note that insertion and deletion costs are all equal in our experiments.)

The "Repair cost" column in Table 2 shows the total repair costs of edit operations to repair all 110 C test programs and all 144 Java test programs using each method. This column shows that [NM] can repair syntax errors with less costs of edit operations than other methods.

The "Failure" column in Table 2 shows the ratios of test programs which each corresponding method fails to repair. This column shows that [CO] fails to find proper repairs in 3.6 % of C programs. As mentioned, [CO] still fails to find proper repairs within reasonable time and space even if N_i and N_d become sufficiently large.

The "< [NM]", "= [NM]" and "> [NM]" columns in Table 2 show the relative numbers of test programs repaired with less costs than [NM], with the same costs as [NM], and with greater costs than [NM], respectively. These columns show that [NM] can repair syntax errors with better repair quality than [CE], [KC] and [CO].

Figures 27, 28 show the repair times using [NM] and [CO] with the same parameter settings as the previous experiment. They show that [NM] with the parameters of $N_i = 100$, $N_d = 100$ and $N_t = 30$ is as efficient as [CO] with the parameters of $N_i = 4$, $N_d = 3$, $N_t = 10$ and $N = 3$. The x-axes and y-axes are all log-scaled in order to show each point clearly. The test programs that [CO] fails to repair are represented as the X-marks in Fig. 27.

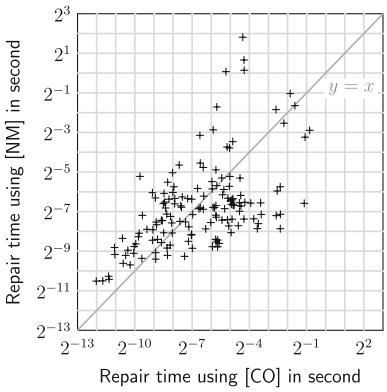


Fig. 28 Repair times for Java programs

6 Conclusions

In this article, we have presented a local LR error repair method that finds high-quality repairs efficiently. The method improves on existing works both in repair efficiency and in repair quality, in that it repairs syntax errors quickly by using the A* algorithm and it enhances the repair quality by supporting unrestricted shifts, as well as insertions and deletions, to repair invalid input strings. This method also accelerates repair process and reduces memory use by removing unproductive and redundant configurations. Experimental results show that the new method is excellent to repair syntax errors with respect to efficiency and quality.

In evaluating error repair methods, the difficulty is how to obtain unbiased test programs and how to evaluate the repair quality. It is very crucial to obtain unbiased test programs because biased test programs tend to lead to biased results. In existing works, evaluating the repair quality has been dependent on a rather subjective evaluation of the experimenters. We believe that our test programs and evaluation method are more persuasive than those in existing works although they may not be perfect.

Appendix

Theorem 1

$$
\begin{aligned}
&\mathrm{Follow}_a(q_1 \ldots q_i) \\
&= \{xa \in T^* \mid q_i \overset{\alpha}{\rightsquigarrow} q \xrightarrow{a} q' \textit{ in LR machine, } \alpha a \Rightarrow^* xa\} \\
&\quad \cup \bigcup_{[A \to \alpha \bullet \beta] \in \mathrm{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \mathrm{Follow}_a(q_1 \ldots q_{i-|\alpha|} q) \\
&\qquad \textit{where } \mathrm{action}[q_{i-|\alpha|}, A] = \texttt{goto}\ q
\end{aligned}
$$

Proof From Definition 1,

$$
\mathrm{Follow}_a(q_1 \ldots q_i) = \{xa \mid (q_1 \ldots q_i, xay\$) \xrightarrow{\mathrm{LR}}{}^{*} (q_1 \ldots q_j, y\$)\}.
$$

$\mathrm{Follow}_a(q_1 \ldots q_i)$ can also be expressed as

$$
\{xa \mid xay \in \mathrm{Follow}(q_1 \ldots q_i)\}.
$$

On the other hand, from Lemma 3,

$$\begin{aligned}&\text{Follow}(q_1 \dots q_i)\\ &= \bigcup_{[A\to\alpha\bullet\beta]\,\in\,\text{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \dots q_{i-|\alpha|}q)\\ &\qquad\qquad \text{where } q_{i-|\alpha|} \xrightarrow{A} q.\end{aligned}$$

Thus, $\text{Follow}_a(q_1 \dots q_i)$ can be rewritten as

$$\begin{aligned}&\text{Follow}_a(q_1 \dots q_i)\\ &= \{xa \mid xay \in \text{Follow}(q_1 \dots q_i)\}\\ &= \bigcup_{[A\to\alpha\bullet\beta]\,\in\,\text{kernel}(q_i)} \{xa \in T^* \mid \beta \Rightarrow^* xay\}\\ &\cup \bigcup_{[A\to\alpha\bullet\beta]\,\in\,\text{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}_a(q_1 \dots q_{i-|\alpha|}q)\\ &\qquad\qquad \text{where } q_{i-|\alpha|} \xrightarrow{A} q.\end{aligned}$$

Then, from Lemma 4, we conclude that

$$\begin{aligned}&\text{Follow}_a(q_1 \dots q_i)\\ &= \{xa \in T^* \mid q \overset{\gamma a}{\rightsquigarrow} q' \text{ in LR machine, } \gamma a \Rightarrow^* xa\}\\ &\cup \bigcup_{[A\to\alpha\bullet\beta]\,\in\,\text{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}_a(q_1 \dots q_{i-|\alpha|}q)\\ &\qquad\qquad \text{where } q_{i-|\alpha|} \xrightarrow{A} q.\end{aligned}$$

Lemma 1

$$\begin{aligned}&\text{Follow}(q_1 \dots q_i)\\ &= \bigcup_{[A\to\alpha\bullet\beta]\in q_i} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \dots q_{i-|\alpha|}q) \textit{ where } q_{i-|\alpha|} \xrightarrow{A} q\end{aligned}$$

Proof By Definition 1,

$$\text{Follow}(q_1 \dots q_i) = \{x \mid (q_1 \dots q_i, x\$) \xrightarrow{\text{LR}}{}^* (q_1 \dots q_j, \$), \text{action}[q_j, \$] = \text{accept}\}.$$

Then, for any x in $\text{Follow}(q_1 \dots q_i)$, there exists an item $[A \to \alpha \bullet \beta]$ in q_i such that $\beta \Rightarrow^* y$ and $x = yz$ for some terminal strings y and z, satisfying

$$\begin{aligned}&(q_1 \dots q_i, x\$)\\ &= (q_1 \dots q_i, yz\$)\\ &\xrightarrow{\text{LR}}{}^* (q_1 \dots q_{i+|\beta|}, z\$)\\ &\xrightarrow{\text{LR}} (q_1 \dots q_{i-|\alpha|}q, z\$) \quad \text{reducing } A \to \alpha\beta \text{ and } q_{i-|\alpha|} \xrightarrow{A} q\\ &\xrightarrow{\text{LR}}{}^* (q_1 \dots q_j, \$) \quad \text{and} \quad \text{action}[q_j, \$] = \text{accept}.\end{aligned}$$

Thus, for any x in $\text{Follow}(q_1 \dots q_i)$,

$$x \in \{w \in T^* \mid \beta \Rightarrow^* w\} \cdot \text{Follow}(q_1 \dots q_{i-|\alpha|}q)$$

for some $[A \to \alpha \bullet \beta] \in q_i$ and $q_{i-|\alpha|} \xrightarrow{A} q$. We conclude that

$$\begin{aligned}&\text{Follow}(q_1 \dots q_i)\\ &= \bigcup_{[A\to\alpha\bullet\beta]\in q_i} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \dots q_{i-|\alpha|}q) \text{ where } q_{i-|\alpha|} \xrightarrow{A} q.\end{aligned}$$

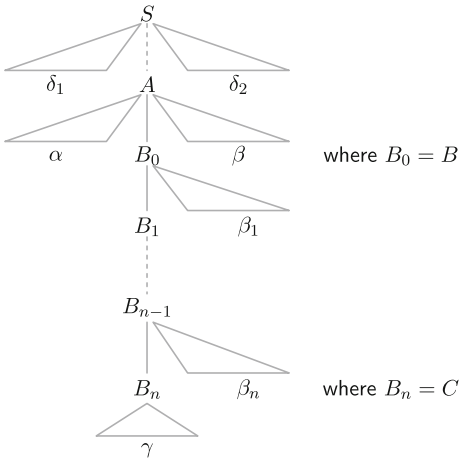
Lemma 2 *For any non-kernel item* $[C \to \bullet\gamma]$ *in* q_i*, the set*

$$\{x \in T^* \mid \gamma \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_i q) \textit{ where } q_i \xrightarrow{C} q$$

is included in

$$\bigcup_{\substack{[A \to \alpha\bullet\beta] \\ \in \text{kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q') \textit{ where } q_{i-|\alpha|} \xrightarrow{A} q'.$$

Proof For any non-kernel item $[C \to \bullet\gamma]$ in q_i, there always exists at least one kernel item $[A \to \alpha \bullet B\beta]$ in q_i satisfying the following derivation tree.



Thus, for any $w \in \{x \in T^* \mid \gamma \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_i q)$, there exists a kernel item $[A \to \alpha \bullet B\beta]$ in q_i satisfying the above derivation relation such that

$$w \in \{x \in T^* \mid \gamma\beta_n \ldots \beta_1 \beta\delta_2 \Rightarrow^* x\}.$$

This relation can be re-expressed as

$$w \in \{x \in T^* \mid B\beta\delta_2 \Rightarrow^* x\}$$

since $B \Rightarrow^* \gamma\beta_n \ldots \beta_1$, and also as

$$w \in \{x \in T^* \mid B\beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q') \text{ where } q_{i-|\alpha|} \xrightarrow{A} q'$$

since $\{x \in T^* \mid \delta_2 \Rightarrow^* x\} \subseteq \text{Follow}(q_1 \ldots q_{i-|\alpha|} q')$. Hence,

$$w \in \bigcup_{\substack{[A \to \alpha\bullet\beta] \\ \in \text{kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q') \text{ where } q_{i-|\alpha|} \xrightarrow{A} q'.$$

It follows that, for any non-kernel item $[C \to \bullet\gamma]$ in q_i, the set

$$\{x \in T^* \mid \gamma \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_i q) \text{ where } q_i \xrightarrow{C} q$$

is included in

$$\bigcup_{\substack{[A\to\alpha\bullet\beta]\\ \in \text{kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q') \text{ where } q_{i-|\alpha|} \xrightarrow{A} q'.$$

Lemma 3

$$\begin{aligned}&\text{Follow}(q_1 \ldots q_i)\\ &= \bigcup_{[A\to\alpha\bullet\beta] \in \text{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q)\\ &\qquad\qquad \text{where } q_{i-|\alpha|} \xrightarrow{A} q\end{aligned}$$

Proof By Lemma 1,

$$\text{Follow}(q_1 \ldots q_i) = \bigcup_{[A\to\alpha\bullet\beta]\in q_i} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q) \text{ where } q_{i-|\alpha|} \xrightarrow{A} q.$$

As $q_i = \text{kernel}(q_i) \cup \text{non-kernel}(q_i)$, $\text{Follow}(q_1 \ldots q_i)$ can be re-expressed as follows.

$$\begin{aligned}&\text{Follow}(q_1 \ldots q_i)\\ &= \bigcup_{\substack{[A\to\alpha\bullet\beta]\in\\ \text{kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q) \text{ where } q_{i-|\alpha|} \xrightarrow{A} q\\ &\cup \bigcup_{\substack{[A\to\bullet\beta]\in\\ \text{non-kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_i q) \text{ where } q_i \xrightarrow{A} q\end{aligned}$$

By Lemma 2,

$$\bigcup_{\substack{[A\to\bullet\beta]\in\\ \text{non-kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_i q) \text{ where } q_i \xrightarrow{A} q$$

in included in

$$\bigcup_{[A\to\alpha\bullet\beta]\in\text{kernel}(q_i)} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q) \text{ where } q_{i-|\alpha|} \xrightarrow{A} q.$$

Hence, we conclude that

$$\text{Follow}(q_1 \ldots q_i) = \bigcup_{\substack{[A\to\alpha\bullet\beta]\\ \in \text{kernel}(q_i)}} \{x \in T^* \mid \beta \Rightarrow^* x\} \cdot \text{Follow}(q_1 \ldots q_{i-|\alpha|} q) \text{ where } q_{i-|\alpha|} \xrightarrow{A} q.$$

Lemma 4 *For a given terminal symbol a,*

$$\begin{aligned}&\bigcup_{[A\to\alpha\bullet\beta] \in \text{kernel}(q)} \{xa \in T^* \mid \beta \Rightarrow^* xay\}\\ &\qquad = \{xa \in T^* \mid q \overset{\gamma a}{\rightsquigarrow} q' \textit{ in LR machine, } \gamma a \Rightarrow^* xa\}.\end{aligned}$$

Proof We will prove this lemma by showing two sets are included in each other as follows.

(1) There exists some viable prefix δ leading to state q in LR machine. If a kernel item $[A \to \alpha \bullet \beta]$ is in q, then there is a derivation relation as follows.

$$S \Rightarrow_r^* \delta_1 Az \Rightarrow_r \delta_1\alpha\beta z = \delta\beta z$$

for some viable prefix δ_1 and $z \in T^*$. Meanwhile, if $\beta \Rightarrow_r^* xay \in T^*$, then there exists a vocabulary string γa such that $\beta \Rightarrow_r^* \gamma ay \Rightarrow_r^* xay$. Thus, there exists a derivation relation as

$$S \Rightarrow_r^* \delta\beta z \Rightarrow_r^* \delta\gamma ayz \Rightarrow_r^* \delta xayz$$

meaning that there exists a viable prefix $\delta\gamma a$ and $\gamma a \Rightarrow^* xa$. Suppose that $\delta\gamma a$ leads to a state q' in the LR machine. As δ leads to a state q, there exists a path $q \overset{\gamma a}{\rightsquigarrow} q'$ in the LR machine and $\gamma a \Rightarrow^* xa$. Hence, we conclude that

$$\bigcup_{[A\to\alpha\bullet\beta]\in \text{kernel}\,(q)} \{xa \in T^* \mid \beta \Rightarrow^* xay\} \subseteq \{xa \in T^* \mid q \overset{\gamma a}{\rightsquigarrow} q' \text{ in LR machine},\ \gamma a \Rightarrow^* xa\}.$$

(2) Lemma 5 shows that if there is a path $q \overset{\gamma a}{\rightsquigarrow} q'$ for some state q in LR machine, then there exists a kernel item $[A \to \alpha \bullet \beta]$ in q such that $\beta \Rightarrow_r^* \gamma ax$ for some $x \in T^*$. From Lemma 5, we know

$$\{\gamma a \mid q \overset{\gamma a}{\rightsquigarrow} q' \text{ in LR machine}\} \subseteq \bigcup_{[A\to\alpha\bullet\beta]\in \text{kernel}\,(q)} \{\gamma a \mid \beta \Rightarrow_r^* \gamma ax,\ ax \in T^*\}.$$

Hence,

$$\{xa \in T^* \mid q \overset{\gamma a}{\rightsquigarrow} q' \text{ in LR machine},\ \gamma a \Rightarrow^* xa\} \subseteq \bigcup_{[A\to\alpha\bullet\beta]\in \text{kernel}\,(q)} \{xa \in T^* \mid \beta \Rightarrow^* xay\}.$$

From (1) and (2), we conclude that

$$\bigcup_{[A\to\alpha\bullet\beta]\in \text{kernel}(q)} \{xa \in T^* \mid \beta \Rightarrow^* xay\} = \{xa \in T^* \mid q \overset{\gamma a}{\rightsquigarrow} q' \text{ in LR machine},\ \gamma a \Rightarrow^* xa\}.$$

Lemma 5 *If there is a path* $q \overset{\gamma}{\rightsquigarrow} q'$ $(|\gamma| > 0)$ *for some LR state* q *in LR-machine, then there exists a kernel item* $[A \to \alpha \bullet \beta]$ *in* q *such that* $\beta \Rightarrow_r^* \gamma x$ *for some* $x \in T^*$.

Proof We will prove this lemma by induction on the length of γ.

(1) Suppose that $|\gamma| = 1$ and $\gamma = X$.
Then, q has some item $[B \to \delta_1 \bullet X\delta_2]$.

- If $[B \to \delta_1 \bullet X\delta_2]$ is a kernel item in q, then it is evident that there exists a kernel item $[B \to \delta_1 \bullet X\delta_2]$ in q such that $X\delta_2 \Rightarrow_r^* Xx$ for some $x \in T^*$.
- If $[B \to \delta_1 \bullet X\delta_2]$ is a non-kernel item, then $\delta_1 = \varepsilon$ and there is some kernel item $[A \to \alpha \bullet \beta]$ in q such that $\beta \Rightarrow_r^* X\delta_2 y$ for some $y \in T^*$. Hence, there is some kernel item $[A \to \alpha \bullet \beta]$ in q such that $\beta \Rightarrow_r^* X\delta_2 y \Rightarrow_r^* Xxy$ for some $x, y \in T^*$.

(2) Suppose that $|\gamma| > 1$ and $\gamma = X\gamma'$.

Let $q \xrightarrow{X} q'' \overset{\gamma'}{\rightsquigarrow} q'$ for some state q''. Then, by the inductive hypothesis, there exists a kernel item $[A \to \alpha X \bullet \beta]$ in q'' such that $\beta \Rightarrow_r^* \gamma' x$ for some $x \in T^*$. (Note that any kernel item in q'' is of the form of $[A \to \alpha X \bullet \beta]$ because $q \xrightarrow{X} q''$.) From $q \xrightarrow{X} q''$, we know that there is an item $[A \to \alpha \bullet X\beta]$ in q.

- If $[A \to \alpha \bullet X\beta]$ is a kernel item in q, then it is clear that there exists a kernel item $[A \to \alpha \bullet X\beta]$ in q such that $X\beta \Rightarrow_r^* X\gamma' x$ for some $x \in T^*$. (Note that $\beta \Rightarrow_r^* \gamma' x$ for some $x \in T^*$.)
- If $[A \to \alpha \bullet X\beta]$ is a non-kernel item, then $\alpha = \varepsilon$ and there is some kernel item $[A' \to \alpha' \bullet \beta']$ in q such that $\beta' \Rightarrow_r^* X\beta y$ for some $y \in T^*$. Hence, there is some kernel item $[A' \to \alpha' \bullet \beta']$ in q such that

$$\beta' \Rightarrow_r^* X\beta y \Rightarrow_r^* X\gamma' xy$$

for some $x, y \in T^*$. (Note that $\beta \Rightarrow_r^* \gamma' x$ for some $x \in T^*$.)

From (1) and (2), we conclude that this lemma is true.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison Wesley, Reading (1986)
2. Bertsch, E.: An observation on suffix redundancy in LL(1) error repair. Acta Inf. **33**(7), 631–639 (1996)
3. Bertsch, E., Nederhof, M.J.: On failure of the pruning technique in error repair in shift-reduce parsers. ACM Trans. Program. Lang. Syst. **21**(1), 1–10 (1999)
4. Cerecke, C.: Repairing syntax errors in LR-based parsers. In: The Twenty-Fifth Australasian Computer Science Conference, ACSC'02, **4**, pp. 17–22. ACM, Melbourne, Australia (2002)
5. Corchuelo, R., Perez, J.A., Ruiz, A., Toro, M.: Repairing syntax errors in LR parsers. ACM Trans. Program. Lang. Syst. **24**(6), 698–710 (2002)
6. Dain, J.A.: A practical minimum distance method for syntax error handling. Comp. Lang. **20**(4), 239–252 (1994)
7. Dion, B.A.: Locally Least-Cost Error Correctors for Context-Free and Context-Sensitive Parsers, vol. 1. UMI Research Press, Ann Arbor (1982)
8. Fischer, C., Dion, B.A., Mauney, J.: A locally least-cost LR-error corrector. Tech. Rep. 363, Computer Science Dept., Univ. of Wisconsin (1979)
9. Fischer, C.N., Mauney, J.: A simple, fast, and effective LL(1) error repair algorithm. Acta Inf. **29**(2), 109–120 (1992)
10. Fischer, C., Milton, D., Quiring, S.: Efficient LL(1) error correction and recovery using only insertions. Acta Inf. **13**(3), 141–154 (1980)
11. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **SSC-4**(2), 100–1007 (1968)
12. Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to "A formal basis for the heuristic determination of minimum cost paths". SIGART Bull. **37**, 28–29 (1972)
13. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, vol. 1, 2nd edn. Prentice Hall Inc., Englewood Cliffs (1988)
14. Kim, I.S., Choe, K.M.: Error repair with validation in LR-based parsing. ACM Trans. Program. Lang. Syst. **23**(4), 451–471 (2001)
15. Mauney, J., Fischer, C.N.: A forward move algorithm for LL and LR parsers. SIGPLAN Notice **17**(6), 77–89 (1982)
16. McKenzie, B.J., YeatMan, C., Vere, L.D.: Error repair in shift-reduce parsers. ACM Trans. Program. Lang. Syst. **17**(4), 672–689 (1995)
17. Nilsson, N.J.: Principles of Artificial Intelligence. Morgan Kaufmann Publishers, Massachusetts (1993)
18. Pennello, T.J., De Remer, F.: A forward move algorithm for LR error recovery. In: Conference Record of the 5th Annual ACM symposium of Principles of Programming Languages. ACM, Tucson, Arizona, pp. 241–254 (1978)

19. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Pearson Educ. (2003). http://portal.acm.org/citation.cfm?id=773294
20. Sippu, S., Soisalon-Soinnen, E.: Parsing Theory: LR(k) and LL(k) Parsing, vol. 2. Springer, Berlin (1990)
21. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)
22. Winston, P.H.: Artificial Intelligence, 3rd edn. Addison Wesley, Reading (1992)