Syntax Repair as Language Intersection

ANONYMOUS AUTHOR(S)

We introduce a new technique for correcting syntax errors in arbitrary context-free languages. Our work addresses the problem of syntax error correction, which we solve by defining a finite language that provably generates every repair within a certain edit distance. To do this, we adapt the Bar-Hillel construction from formal languages, guaranteeing this language is sound and complete with respect to a programming language's grammar. This technique also admits a polylogarithmic time algorithm for deciding intersection nonemptiness between CFLs and acyclic NFAs, the first of its kind in the parsing literature.

1 INTRODUCTION

When programming, one invariably encounters a recurring scenario in which the editor occupies an unparseable state. Faced with this predicament, programmers must spend time to locate and repair the error before proceeding. In the following paper, we propose to solve this problem automatically by generating a list of candidate repairs which contains with high probability the true repair, assuming this repair differs by no more than a few edits from the broken source code.

Prior research on syntax repair can be classified into exact and approximate methods. In the former, rule-based methods are used to locate a suitable alternative. While appealing for their interpretability and well-understood algorithmic properties, these methods are too weak to model the full distribution of natural source code and must rely on relatively brittle heuristics.

In the latter case, the set of all strings is typically used as the sample space for a distribution whose parameters are learned from a dataset of pairwise errors and fixes. Though statistically more robust, large language models typically use approximate inference and thus require some form of postprocessing or rejection sampling to ensure the generated results conform to the grammar.

The primary shortcoming with both of these approaches is they generate too few repairs. Even if the model in question guarantees grammatical soundness or has good statistical generalization properties, it is likely to miss the intended repair in the presence of ambiguity or when there are many candidates from which to choose. Note however, that most syntax errors need relatively minor alterations to repair, of which there are only a finite number of possibilities to consider.

Thus we arrive at the core problem this paper aims to solve: how do we efficiently recover the most probable repairs in close proximity to a syntactically broken code snippet? To address this problem, we propose to exhaustively evaluate every repair within a fixed edit distance.

Our algorithm first constructs an automaton representing all possible strings within a certain edit distance. This is used to compute a matrix denoting all valid repairs in the programming language and edit distance, then we construct a regular expression for that language. Finally, this regular expression is decoded using a fast pretrained statistical model to produce a finite list, which is then reranked and truncated to obtain the final repairs. The full pipeline is depicted in Fig. 1.

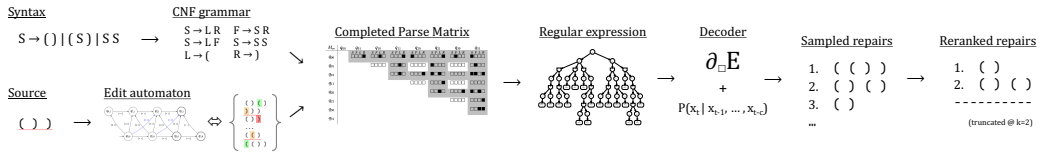


Fig. 1. Simplified dataflow. Given a grammar and broken code fragment, we create an automaton generating the language of small edits, then construct a regular expression representing the intersection of the two languages. This regular expression can be decoded to produce the final list of repairs.

To operationalize this technique, we design, develop and benchmark a new developer tool for syntax repair. This tool makes aggressive use of communication-free parallelism, making it readily executable by off-the-shelf GPU and SIMD co-processors. We provide a reference implementation of our tool on the WebGPU platform and show these computational resources, which typically sit idle during text editing, can be profitably used to accelerate real-time program repair.

Finally, we evaluate our approach on a dataset of human syntax errors and fixes fewer than five lexical edits and shorter than 120 tokens, large enough to fit a few lines of source code in realistic programming languages. Our work shows this technique is highly effective at predicting the true repair across a dataset of Python source code, on average 5x more accurately than previous state of the art methods at comparable latency and compute thresholds.

2 BACKGROUND

Recall that a CFG, $\mathcal{G} = \langle \Sigma, V, P, S \rangle$, is a quadruple consisting of terminals (Σ), nonterminals (V), productions $\left(P\colon V \longrightarrow (V \mid \Sigma)^+\right)$, and a start symbol, ($S$). Every CFG is reducible to so-called *Chomsky Normal Form* [16], $P'\colon V \longrightarrow (V^2 \mid \Sigma)$, where every production is either (1) a binary production $w \longrightarrow xz$, or (2) a unit production $w \longrightarrow t$, where $w, x, z : V$ and $t : \Sigma$. For example:

$$G = \left\{ S \longrightarrow S\,S \mid (\,S\,) \mid (\,) \right\} \Longrightarrow G' = \left\{ S \longrightarrow Q\,R \mid S\,S \mid L\,R, \quad R \longrightarrow), \quad L \longrightarrow (, \quad Q \longrightarrow L\,S \right\}$$

Likewise, a finite state automaton (FSA) is a quintuple $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, and $I, F \subseteq Q$ are the set of initial and final states, respectively. These generally come in two varieties, deterministic and nondeterministic depending on whether or not δ maps each pair $\langle q, s \rangle$ to a unique q'.

There is an equivalent characterization of the regular languages via an inductively defined datatype, which is often more elegant than FSAs to work with. Consider the generalized regular expression (GRE or REG) fragment containing concatenation, conjunction and disjunction:

Definition 2.1 (Generalized regular expression). Let e be an expression defined by the grammar:

$$e \longrightarrow \varnothing \mid \varepsilon \mid \Sigma \mid e \cdot e \mid e \vee e \mid e \wedge e$$

Semantically, we can interpret these expressions as denoting regular languages:

$$\begin{aligned} \mathcal{L}(\quad \varnothing \quad) &= \varnothing & \mathcal{L}(\quad x \cdot z \quad) &= \mathcal{L}(x) \circ \mathcal{L}(z)^{1} \\ \mathcal{L}(\quad \varepsilon \quad) &= \{\varepsilon\} & \mathcal{L}(\quad x \vee z \quad) &= \mathcal{L}(x) \cup \mathcal{L}(z) \\ \mathcal{L}(\quad a \quad) &= \{a\} & \mathcal{L}(\quad x \wedge z \quad) &= \mathcal{L}(x) \cap \mathcal{L}(z) \end{aligned}$$

Brzozowski [11] introduces an operator, ∂, which lets us quotient a language by some prefix,

Definition 2.2 (Brzozowski, 1964). To compute the quotient $\partial_a(L) = \{b \mid ab \in L\}$, we:

$$\begin{aligned} \partial_a(\quad \varnothing \quad) &= \varnothing & \delta(\quad \varnothing \quad) &= \varnothing \\ \partial_a(\quad \varepsilon \quad) &= \varnothing & \delta(\quad \varepsilon \quad) &= \varepsilon \\ \partial_a(\quad b \quad) &= \begin{cases} \varepsilon & \text{if } a = b \\ \varnothing & \text{if } a \neq b \end{cases} & \delta(\quad a \quad) &= \varnothing \\ \partial_a(\quad x \cdot z \quad) &= (\partial_a x) \cdot z \vee \delta(x) \cdot \partial_a z & \delta(\quad x \cdot z \quad) &= \delta(x) \wedge \delta(z) \\ \partial_a(\quad x \vee z \quad) &= \partial_a x \vee \partial_a z & \delta(\quad x \vee z \quad) &= \delta(x) \vee \delta(z) \\ \partial_a(\quad x \wedge z \quad) &= \partial_a x \wedge \partial_a z & \delta(\quad x \wedge z \quad) &= \delta(x) \wedge \delta(z) \end{aligned}$$

[1]Where $\mathcal{L}(x) \circ \mathcal{L}(z)$ is defined as $\{a \cdot b \mid a \in \mathcal{L}(x) \wedge b \in \mathcal{L}(z)\}$.

Primarily, this gadget was designed to handle membership queries, for which purpose it has received considerable attention [1, 37, 47, 50] in recent years:

Theorem 2.3 (Recognition). *For any regex e and $\sigma : \Sigma^*$, $\sigma \in \mathcal{L}(e) \iff \varepsilon \in \mathcal{L}(\partial_\sigma e)$, where:*

$$\partial_\sigma(e) : E \to E = \begin{cases} e & \text{if } \sigma = \varepsilon \\ \partial_b(\partial_a e) & \text{if } \sigma = a \cdot b, a \in \Sigma, b \in \Sigma^* \end{cases}$$

Variations on this basic procedure can also be used for functional parsing and regular expression tasks. Less well known, perhaps, is that Brzozowski's derivative can also be used to decode witnesses. We will first focus on the nonempty disjunctive fragment, and define this process in two steps:

Theorem 2.4 (Generation). *For any nonempty $(\varepsilon, \wedge)$-free regex, e, to witness $\sigma \in \mathcal{L}(e)$:*

$$\texttt{follow}(e) : E \to 2^\Sigma = \begin{cases} \{e\} & \text{if } e \in \Sigma \\ \texttt{follow}(x) & \text{if } e = x \cdot z \\ \texttt{follow}(x) \cup \texttt{follow}(z) & \text{if } e = x \vee z \end{cases}$$

$$\texttt{choose}(e) : E \to \Sigma^+ = \begin{cases} e & \text{if } e \in \Sigma \\ \left(s \xleftarrow{\$} \texttt{follow}(e)\right) \cdot \texttt{choose}(\partial_s e) & \text{if } e = x \cdot z \\ \texttt{choose}(e' \xleftarrow{\$} \{x, z\}) & \text{if } e = x \vee z \end{cases}$$

Here, we use the $\xleftarrow{\$}$ operator to denote probabilistic choice, however any deterministic choice function will also suffice to generate a witness. Now we are equipped to handle conjunction.

Recall that every regular language is also context-free a fortiori. So, given an $(\varepsilon, \wedge)$-free regular expression, we can construct an equivalent CFG with productions $P(e)$ as follows:

$$P(e) = \begin{cases} \{S_e \to \sigma \mid \sigma \in \Sigma\} & \text{if } e = \Sigma \\ P(x) \cup P(z) \cup \{S_e \to S_x S_z\} & \text{if } e = x \cdot z \\ P(x) \cup P(z) \cup \{S_e \to S_x, S_e \to S_z\} & \text{if } e = x \vee z \end{cases} \tag{1}$$

where the CFG is $G(e) = \langle V, \Sigma, P(e), S_e \rangle$ with V being all nonterminals in $P(e)$. Therefor, to intersect two regular languages, we can treat one of them as a CFL. Alternatively, we can take the intersection between some truly non-regular CFL (say, a programming language syntax) and a regular language.

Theorem 2.5 (Bar-Hillel, 1961). *For any context-free grammar (CFG), $G = \langle V, \Sigma, P, S \rangle$, and nondeterministic finite automata (NFA), $A = \langle Q, \Sigma, \delta, I, F \rangle$, there exists a CFG $G_\cap = \langle V_\cap, \Sigma_\cap, P_\cap, S_\cap \rangle$ such that $\mathcal{L}(G_\cap) = \mathcal{L}(G) \cap \mathcal{L}(A)$.*

Salomaa [43] introduces a direct, but inefficient construction for the intersection grammar:

Definition 2.6 (Salomaa, 1973). One could construct $G_\cap$ like so,

$$\frac{q \in I \quad r \in F}{(S \to qSr) \in P_\cap}\ \mathcal{S} \qquad \frac{(w \to a) \in P \qquad (q \xrightarrow{a} r) \in \delta}{(qwr \to a) \in P_\cap}\ \uparrow \qquad \frac{(w \to xz) \in P \qquad p, q, r \in Q}{(pwr \to (pxq)(qzr)) \in P_\cap}\ \bowtie$$

however most synthetic productions in $P_\cap$ will be non-generating or unreachable. This method will construct a synthetic production for state pairs which are not even connected by any path, which is clearly excessive. In Sec. 3, we will present a far more efficient construction for the special case when the intersection is finite. But first, let us return to the broader question of syntax repair.

2.1 Informal statement

Assume there exists a transducer from Unicode tokens to grammatical tokens, $\tau : \Sigma_U^* \to \Sigma_G^*$. In the compiler nomenclature τ is called a *lexer* and would typically be regular under mild conditions. In this paper, we do not consider τ and strictly deal with languages over Σ_G^*, or simply Σ^* for brevity.

Now suppose we have a syntax, $\ell \subset \Sigma^*$, containing every acceptable program. A syntax error is an unacceptable string, $\underset{\sim}{\sigma} \notin \ell$, that we wish to repair. We can model syntax repair as a language intersection between a context-free language (CFL) and a regular language. Henceforth, $\underset{\sim}{\sigma}$ will always and only be used to denote a syntactically invalid string whose target language is known.

Given a lexical representation of a broken computer program $\underset{\sim}{\sigma}$ and a grammar G, our goal is to find every valid string σ consistent with the grammar G and within a certain edit distance, d. Consider the language of nearby strings: if intersected with the language of grammatically valid programs, $\mathcal{L}(G)$, the result ($\ell_\cap$) will contain every possible repair within the given edit distance, a subset of which will be natural or statistically probable. If we can locate these repairs then we can map them back into Unicode, adding placeholders for fresh names, numbers, and string literals, then finally apply an off-the-shelf code formatter to display them. Both the preprocessing and the cosmetic postprocessing steps are tangential to this work, in which we confine ourselves to a lexical alphabet.

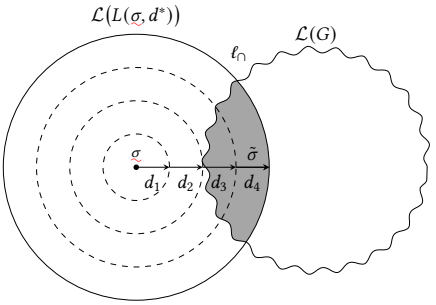


Fig. 2. CFL intersection with the local edit region of a given broken code snippet.

2.2 Formal statement

Let us now restate our informal description of the syntax repair problem in more formal terms.

Definition 2.7 (Bounded Levenshtein-CFL reachability). Given a CFL, ℓ, and an invalid string, $\underset{\sim}{\sigma} : \bar{\ell}$, find every valid string reachable within d edits of $\underset{\sim}{\sigma}$, i.e., letting Δ be the Levenshtein metric and $\mathcal{L}\big(L(\underset{\sim}{\sigma}, d)\big) = \{\sigma' \mid \Delta(\underset{\sim}{\sigma}, \sigma') \leq d\}$ be the Levenshtein d-ball, we seek to find $\ell_\cap = \mathcal{L}\big(L(\underset{\sim}{\sigma}, d)\big) \cap \ell$.

As the admissible set $\ell_\cap$ is typically under-constrained, we want a procedure which surfaces natural and valid repairs over unnatural but valid repairs:

Definition 2.8 (Ranked repair). Given a finite language $\ell_\cap = \mathcal{L}\big(L(\underset{\sim}{\sigma}, d)\big) \cap \ell$ and a probabilistic language model $\mathrm{P}_\theta : \Sigma^* \to [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-k maximum probability repairs under the language model. That is,

$$R(A, P_\theta) = \underset{\boldsymbol{\sigma} \subseteq \ell_\cap, |\boldsymbol{\sigma}| \leq k}{\operatorname{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} \mathrm{P}_\theta(\sigma) \tag{2}$$

A popular approach to ranked repair involves learning a distribution over strings, however this is highly sample-inefficient and generalizes poorly to new languages. Approximating a distribution over Σ^* forces the model to jointly learn syntax and stylometry. Furthermore, even with an extremely efficient approximate sampler for $\sigma \sim \ell_\cap$, due to the size of the languages involved, it would be intractable to sample either ℓ or $\mathcal{L}\big(L(\underset{\sim}{\sigma}, d)\big)$, reject duplicates, then reject unreachable or invalid edits, and completely out of the question to sample $\sigma \sim \Sigma^*$ as do many neural language models.

As we will demonstrate, the ranked repair problem can be factorized into two steps: first exact representation, then decoding. Instead of working with strings, we will explicitly construct a grammar which soundly and completely generates the set $\ell \cap \mathcal{L}\big(L(\underset{\sim}{\sigma}, d)\big)$, then retrieve repairs from its language. By ensuring retrieval is sufficiently precise and exhaustive, maximizing probability over the retrieved set can be achieved with a much simpler, syntax-oblivious language model.

3 METHOD

The key to solving this problem is to treat finite language intersections as matrix exponentiation. We will show a certain correspondence between CFL-REG intersection and a semiring algebra that allows us to quickly decide and witness intersection nonemptiness for finite languages.

Theorem 3.1. *For every CFG, G, and every acyclic NFA (ANFA), A, there exists a decision procedure* $\varphi : CFG \to ANFA \to \mathbb{B}$ *such that* $\varphi(G, A) \models [\mathcal{L}(G) \cap \mathcal{L}(A) \neq \varnothing]$ *which requires* $\mathcal{O}\big(\log^c |Q||V|\big)$ *time using* $\mathcal{O}\big((|Q||V|)^k\big)$ *parallel processors for some* $c, k < \infty$.

Proof. To prove nonemptiness, we must show there exists a path $p \rightsquigarrow r$ in A such that $p \in I, r \in F$ where $p \rightsquigarrow r \vdash S$. At least one two cases must hold for $w \in V$ to parse a given $p \rightsquigarrow r$ pair:

(1) p steps directly to r in which case it suffices to check $\exists a.\big((p \xrightarrow{a} r) \in \delta \wedge (w \to a) \in P\big)$, or,

(2) there is some midpoint $q \in Q$, $p \rightsquigarrow q \rightsquigarrow r$ such that $\exists x, z.\big((w \to xz) \in P \wedge \overbrace{\underbrace{p \rightsquigarrow q}_{x}, \underbrace{q \rightsquigarrow r}_{z}}^{w}\big)$.

This decomposition immediately suggests a dynamic programming solution. Let M be a matrix of type $E^{|Q|\times|Q|\times|V|}$ indexed by Q. Since we assumed δ is acyclic, there exists a topological sort of δ imposing a total order on Q such that M is strictly upper triangular (SUT). Initiate it thusly:

$$M_0[r, c, w] = \bigvee_{a \in \Sigma} \big\{a \mid (w \to a) \in P \wedge (q_r \xrightarrow{a} q_c) \in \delta\big\} \tag{3}$$

Now, our goal will be to find $M = M^2$ such that $\big[M_0[r, c, w] \neq \varnothing\big] \implies \big[M[r, c, w] \neq \varnothing\big]$ under a certain near-semiring. The algebraic operations $\oplus, \otimes : E^{2|V|} \to E^{|V|}$ we will define elementwise:

$$[\ell \oplus r]_w = [\ell_w \vee r_w] \quad \text{and} \quad [\ell \otimes r]_w = \bigvee_{x,z \in V} \big\{\ell_x \cdot r_z \mid (w \to xz) \in P\big\} \tag{4}$$

By slight abuse of notation[2], we will redefine the matrix exponential over this domain as:

$$\exp(M) = \sum_{i=0}^{\infty} M_0^i = \sum_{i=0}^{|Q||V|} M_0^i \text{ (since } M \text{ is SUT.)} \tag{5}$$

While $|Q||V|$ is an upper-bound and $\exp(M)$ may converge sooner, incremental evaluation grows expensive even with unbounded parallelism. Instead, we will employ exponentiation-by-squaring:

$$T(2n) = \begin{cases} M_0, & \text{if } n = 1, \\ T(n) + T(n)^2 & \text{otherwise.} \end{cases} \tag{6}$$

Therefor, the complexity can be reduced to at most $\lceil \log_2 |Q||V| \rceil$ sequential steps in the limit. Finally, we will union all the languages of every state pair deriving S into a new nonterminal, $S_\cap$.

$$S_\cap = \bigvee_{q \in I,\, q' \in F} \exp(M)[q, q', S] \text{ and } \varphi = [S_\cap \neq \varnothing] \tag{7}$$

Note we can check φ before each recurrence of T and escape immediately thereafter in case of nonemptiness. Should that occur, one may simply `choose`$(S_\cap)$ to decode a witness (see Thm. 2.4). In either case, the algorithm provably terminates in $\mathcal{O}(\log^c |Q||V|)$ parallel time for finite c. □

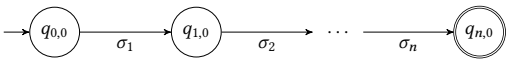
[2]Customarily, there is a $\frac{1}{k!}$ factor to suppress exploding entries, but alas this domain has no multiplicative inverse.

4 EXAMPLES

In this section, we will consider three examples of intersections with finite languages. First, parsing can be viewed as a special case of language intersection with an automaton accepting a single word. Second, completion can be seen as a case of intersection with terminal wildcards in known locations. Thirdly, we consider syntax repair, where we will intersect a language representing all possible edit paths to determine the edit location(s) and fill them with appropriate terminals.

4.1 Recognition as intersection

In the case of ordinary CFL recognition, the automaton accepts just a single word:



Given a CFG, $G' : \mathcal{G}$ in Chomsky Normal Form (CNF), we can construct a recognizer $R : \mathcal{G} \to \Sigma^n \to \mathbb{B}$ for strings $\sigma : \Sigma^n$ as follows. Let 2^V be our domain, 0 be $\varnothing$, $\oplus$ be $\cup$, and $\otimes$ be defined as:

$$X \otimes Z = \left\{ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \right\} \tag{8}$$

If we define $\hat{\sigma}_r = \{w \mid (w \to \sigma_r) \in P\}$, then construct a matrix with nonterminals on the superdiagonal representing each token, $M_0[r+1 = c](G', \sigma) = \hat{\sigma}_r$, the fixpoint $M_{i+1} = M_i + M_i^2$ is uniquely determined by the superdiagonal entries. Omitting the exponentiation-by-squaring detail, the ordinary fixedpoint iteration simply fills successive diagonals:

$$M_0 = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \varnothing & \cdots & \varnothing \\ & \ddots & \ddots & & \\ & & & & \varnothing \\ & & & & \hat{\sigma}_n \\ \varnothing & & & & \varnothing \end{pmatrix} \Rightarrow \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \Lambda & \cdots & \varnothing \\ & \ddots & \ddots & & \\ & & & & \Lambda \\ & & & & \hat{\sigma}_n \\ \varnothing & & & & \varnothing \end{pmatrix} \Rightarrow \ldots \Rightarrow M_\infty = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \Lambda & \cdots & \Lambda^*_\sigma \\ & \ddots & \ddots & & \\ & & & & \Lambda \\ & & & & \hat{\sigma}_n \\ \varnothing & & & & \varnothing \end{pmatrix}$$

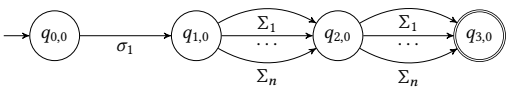
Once the fixpoint M_∞ is attained, the proposition $[S \in \Lambda^*_\sigma]$ decides language membership, i.e., $[\sigma \in \mathcal{L}(G)]$ [3]. So far, this procedure is essentially the textbook CYK algorithm in a linear algebraic notation [24] and a well-established technique in the parsing literature [26].

4.2 Completion as intersection

We can also consider a more general automaton for completing a string with holes, representing edits in fixed locations which can be filled by any terminal, which we call *completion*. In this case, the fixpoint is characterized by a system of language equations, whose solutions are the set of all sentences consistent with the template.

Definition 4.1 (Completion). Let $\underline{\Sigma} = \Sigma \cup \{\_\}$, where _ denotes a hole. We denote $\sqsubseteq : \Sigma^n \times \underline{\Sigma}^n$ as the relation $\{\langle \sigma', \sigma \rangle \mid \sigma_i \in \Sigma \implies \sigma'_i = \sigma_i\}$ and the set of all inhabitants $\{\sigma' : \Sigma^+ \mid \sigma' \sqsubseteq \sigma\}$ as $\mathrm{H}(\sigma)$. Given a *porous string*, $\sigma : \underline{\Sigma}^*$ we seek all syntactically valid inhabitants, i.e., $A(\sigma) = \mathrm{H}(\sigma) \cap \ell$.

Here, the FSA takes a similar shape but can have multiple arcs between adjacent states, e.g.:



[3] Hereinafter, we use Iverson brackets to denote the indicator function of a predicate with free variables, i.e., $[P] \Leftrightarrow \mathbb{1}(P)$.

This corresponds to a template with two holes, $\sigma = 1\ \_\ \_$. Suppose the context-free grammar is $G = \{S \to NON, O \to + \mid \times, N \to 0 \mid 1\}$. This grammar will first be rewritten into CNF as $G' = \{S \to NL, N \to 0 \mid 1, O \to \times \mid +, L \to ON\}$. Using the powerset algebra we just defined, the matrix fixpoint $M' = M + M^2$ can be computed as follows, shown in the leftmost column below:

	2^V	$\mathbb{Z}_2^{\lvert V\rvert}$	$\text{GRE}^{\lvert V\rvert}$
M_0	$\begin{pmatrix} \{N\} & & \\ & \{N,O\} & \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \blacksquare\square\square\square & & \\ & \square\blacksquare\blacksquare\square & \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$ (L N O S)	$\begin{pmatrix} E_{0,1} & & \\ & E_{1,2} & \\ & & E_{2,3} \end{pmatrix}$
M_1	$\begin{pmatrix} \{N\} & \varnothing & \\ & \{N,O\} & \{L\} \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \blacksquare\square\square\square & \square\square\square\square & \\ & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} E_{0,1} & E_{0,2} & \\ & E_{1,2} & E_{1,3} \\ & & E_{2,3} \end{pmatrix}$
$M_2 = M_\infty$	$\begin{pmatrix} \{N\} & \varnothing & \{S\} \\ & \{N,O\} & \{L\} \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \blacksquare\square\square\square & \square\square\square\square & \square\square\square\blacksquare \\ & \square\blacksquare\blacksquare\square & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} E_{0,1} & E_{0,2} & E_{0,3} \\ & E_{1,2} & E_{1,3} \\ & & E_{2,3} \end{pmatrix}$

The same procedure can be translated, without loss of generality, into the bit domain ($\mathbb{Z}_2^{|V|}$) using a lexicographic nonterminal ordering, however M_∞ in both 2^V and $\mathbb{Z}_2^{|V|}$ represents a decision procedure, i.e., $\left[S \in M_\infty[0,3]\right] \Leftrightarrow \left[M_\infty[0,3,3] = \blacksquare\right] \Leftrightarrow \left[A(\sigma) \neq \varnothing\right]$. Since $M_\infty[0,3] = \{S\}$, we know there is at least one $\sigma' \in A(\sigma)$, but neither M_∞ in 2^V or $\mathbb{Z}_2^V$ lets us recover a witness.

To witness $\sigma' \in A(\sigma)$, we can translate the matrix exponential to the GRE domain. We first define $X \boxtimes Z = [X_2 \cdot Z_1, \varnothing, \varnothing, X_1 \cdot Z_0]$ and $X \boxplus Z = [X_i \vee Z_i]_{i \in [0,|V|)}$, mirroring $\oplus, \otimes$ from the powerset domain. Since the unit nonterminals O, N can only occur on the superdiagonal, they may be safely ignored by $\boxtimes$. To solve for M_∞, we proceed by first computing $E_{0,2}, E_{1,3}$:

$$\begin{aligned} E_{0,2} &= E_{0,j} \cdot E_{j,2} = E_{0,1} \boxtimes E_{1,2} \\ &= [L \in E_{0,2}, \varnothing, \varnothing, S \in E_{0,2}] \\ &= [O \in E_{0,1} \cdot N \in E_{1,2}, \varnothing, \varnothing, N \in E_{0,1} \cdot L \in E_{1,2}] \\ &= [E_{0,1,2} \cdot E_{1,2,1}, \varnothing, \varnothing, E_{0,1,1} \cdot E_{1,2,0}] \end{aligned} \qquad \begin{aligned} E_{1,3} &= E_{1,j} \cdot E_{j,3} = E_{1,2} \boxtimes E_{2,3} \\ &= [L \in E_{1,3}, \varnothing, \varnothing, S \in E_{1,3}] \\ &= [O \in E_{1,2} \cdot N \in E_{2,3}, \varnothing, \varnothing, N \in E_{1,2} \cdot L \in E_{2,3}] \\ &= [E_{1,2,2} \cdot E_{2,3,1}, \varnothing, \varnothing, E_{1,2,1} \cdot E_{2,3,0}] \end{aligned}$$

Now we solve for the corner entry $E_{0,3}$ by dotting the first row and last column, which yields:

$$\begin{aligned} E_{0,3} &= E_{0,j} \cdot E_{j,3} = (E_{0,1} \boxtimes E_{1,3}) \boxplus (E_{0,2} \boxtimes E_{2,3}) \\ &= [E_{0,1,2} \cdot E_{1,3,1} \vee E_{0,2,2} \cdot E_{2,3,1}, \varnothing, \varnothing, E_{0,1,1} \cdot E_{1,3,0} \vee E_{0,2,1} \cdot E_{2,3,0}] \end{aligned}$$

Since we only care about $E_{0,3,3} \Leftrightarrow [S \in E_{0,3}]$, we can ignore the first three entries and solve for:

$$\begin{aligned} E_{0,3,3} &= E_{0,1,1} \cdot E_{1,3,0} \vee E_{0,2,1} \cdot E_{2,3,0} \\ &= E_{0,1,1} \cdot (E_{1,2,2} \cdot E_{2,3,1}) \vee E_{0,2,1} \cdot \varnothing \\ &= E_{0,1,1} \cdot E_{1,2,2} \cdot E_{2,3,1} \big(= [N \in E_{0,1}] \cdot [O \in E_{1,2}] \cdot [N \in E_{2,3}] \big) \\ &= 1 \cdot \{+, \times\} \cdot \{0, 1\} \end{aligned}$$

Finally, to recover a witness, we can simply $\texttt{choose}\big(1 \cdot \{+, \times\} \cdot \{0, 1\}\big)$.

4.3 Repair as intersection

Now, we are ready to consider the general case of syntax repair, in which case the edit locations are not localized but can occur anywhere inside the snippet. In this case, we construct a lattice of all possible edit paths up to a fixed distance. This structure is called a Levenshtein automaton.

As the original construction defined by Schultz and Mihov [44] contains cycles and ε-transitions, we propose a variant which is ε-free and acyclic. Furthermore, we adopt a nominal form which supports infinite alphabets and simplifies the description to follow. Illustrated in Fig. 3 is an example of a small Levenshtein automaton recognizing $\mathcal{L}\big(L(\sigma : \Sigma^5, 3)\big)$. Unlabeled arcs accept any terminal from the alphabet, Σ. Equivalently, this transition system can be viewed as a kind of proof system within an unlabeled lattice. The following construction is equivalent to Schultz and Mihov's original Levenshtein automaton, but is more amenable to our purposes as it does not any contain ε-arcs, and instead uses skip connections to recognize consecutive deletions of varying lengths.

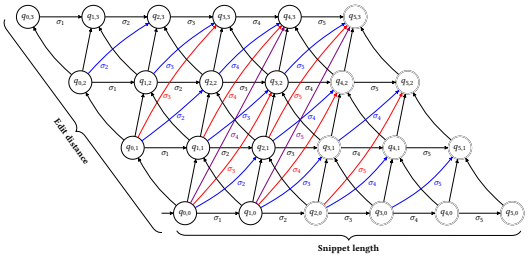


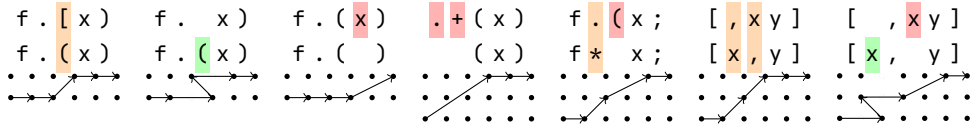
Fig. 3. Levenshtein NFA recognizing $\mathcal{L}\big(L(\sigma : \Sigma^5, 3)\big)$.

$$\frac{s \in \Sigma \quad i \in [0,n] \quad j \in [1,d_{\max}]}{(q_{i,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \qquad \frac{s \in \Sigma \quad i \in [1,n] \quad j \in [1,d_{\max}]}{(q_{i-1,j-1} \xrightarrow{s} q_{i,j}) \in \delta}$$

$$\frac{i \in [1,n] \quad j \in [0,d_{\max}]}{(q_{i-1,j} \xrightarrow{\sigma_i} q_{i,j}) \in \delta} \qquad \frac{d \in [1,d_{\max}] \quad i \in [d+1,n] \quad j \in [d,d_{\max}]}{(q_{i-d-1,j-d} \xrightarrow{\sigma_i} q_{i,j}) \in \delta}$$

$$\frac{}{q_{0,0} \in I}\ \textsc{Init} \qquad \frac{q_{i,j} \in Q \qquad |n-i+j| \leq d_{\max}}{q_{i,j} \in F}\ \textsc{Done}$$

Each type of arc plays a specific role. ↖ handles insertions, ↗ handles substitutions and ⇗ handles deletions of one or more terminals. Let us consider some illustrative cases.



Note that the same patch can have multiple Levenshtein alignments. Done constructs the final states, which are all states accepting strings σ' whose Levenshtein distance $\Delta(\sigma, \sigma') \leq d_{\max}$.

To avoid creating a parallel bundle of arcs for each insertion and substitution point, we instead decorate each arc with a nominal predicate, accepting or rejecting σ_i. To distinguish this nominal variant from the original construction, we highlight the modified rules in orange below.

$$\frac{i \in [0,n] \quad j \in [1,d_{\max}]}{(q_{i,j-1} \xrightarrow{[\neq\sigma_{i+1}]} q_{i,j}) \in \delta} \qquad \frac{i \in [1,n] \quad j \in [1,d_{\max}]}{(q_{i-1,j-1} \xrightarrow{[\neq\sigma_i]} q_{i,j}) \in \delta}$$

$$\frac{i \in [1,n] \quad j \in [0,d_{\max}]}{(q_{i-1,j} \xrightarrow{[=\sigma_i]} q_{i,j}) \in \delta} \qquad \frac{d \in [1,d_{\max}] \quad i \in [d+1,n] \quad j \in [d,d_{\max}]}{(q_{i-d-1,j-d} \xrightarrow{[=\sigma_i]} q_{i,j}) \in \delta}$$

Nominalizing the NFA eliminates the creation of $e = 2(|\Sigma| - 1) \cdot |\sigma| \cdot d_{\max}$ unnecessary arcs over the entire Levenshtein automaton and drastically reduces the representation size, but does not affect the underlying semantics. Thus, it is important to first nominalize the automaton before proceeding to avoid a large blowup in the intermediate grammar.

As a concrete example, suppose we have the string, $\sigma = \texttt{())}$ and wish to balance the parentheses. We will initially have the Levenshtein automaton, A, depicted in Fig. 4. To check for non-emptiness, we will perform the following procedure. Suppose we have a CNF CFG, $G' = \{S \to LR, S \to LF, S \to SS, F \to SR, L \to (, R \to)\}$ and let us assume an ordering of S, F, L, R on V.

First, we need to order the automata states by increasing longest-path distance from q_0. One approach would be to topologically sort the adjacency matrix. While some form of sorting is unavoidable for arbitrary ANFAs, if we know ahead of time that our structure is a Levenshtein automaton, we can simply enumerate its state space by increasing Manhattan distance from the origin. So, a valid ordering on Q would be $q_{00}, q_{01}, q_{10}, q_{11}, q_{20}, q_{21}, q_{30}, q_{31}$. Now, we want to compute whether $[\mathcal{L}(G') \cap \mathcal{L}(A) \neq \varnothing]$.

Under such an ordering, the adjacency matrix takes an upper triangular form and becomes the template for the initial parse chart, M_0 (Fig. 7). Each entry of this chart corresponds to a vector of expressions $E^{|V|}$ with at least one expression denoting a nonempty language. Likewise, the reachability matrix signifies a subset of state pairs which can participate in the language intersection. The adjacency and reachability matrices will always cover the expression vectors of the initial and final parse charts, respectively. In other words, we may safely ignore absent $\langle q, q' \rangle$ pairs in the reachability matrix, as these state pairs definitely cannot participate in the intersection.

From the reachability matrix we can construct the parse chart via matrix exponentiation. We note that n-step reachability constraints n-step parseability, i.e., $\sum_{i=0}^{n} A^i[q, q'] = \square \vdash M_n[q, q', v] = \square$, thus we can avoid substantial work via memoization. In this example, since $M_\infty[q_{00}, q_{31}, S] = \blacksquare$, this implies that $\mathcal{L}(A) \cap \mathcal{L}(G') \neq \varnothing$, hence $\mathrm{LED}(\sigma, G) = 1$. Using the same matrix, we will then perform a second pass to construct regular expressions representing finite languages for each nonempty constituent. Once again, we can skip $\langle q, q', v \rangle$ entries when $M_\infty[q, q', v] = \square$ to hasten convergence.

Just as before, we will define $\boxplus, \boxtimes$ over GRE vectors, where $X \boxtimes Z = [X_x \cdot Z_z \mid (w \to xz) \in P]_{w \in V}$ and $X \boxplus Z = [X_x \vee Z_z]_{w \in V}$. Finally, we will repeat the matrix exponential, using M_∞ in the binary domain as a guide. This allows us to construct the regular expression tree for $G_\cap = q_{00} S q_{20} \vee q_{00} S q_{31}$ shown in Fig. 9. Once this regex is constructed, decoding becomes simply a matter of invoking $\texttt{choose}(G_\cap)$ to produce concrete repairs. In this case, there are only three in total, `()`, `()()` and `(())`.

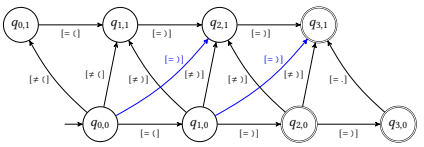


Fig. 4. Simple Levenshtein automaton.

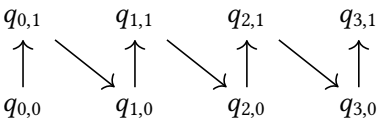


Fig. 5. Pairing function over $\mathcal{L}\big(L(\sigma : \Sigma^3, 1)\big)$.

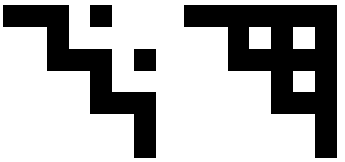


Fig. 6. Adjacency and reachability matrix.

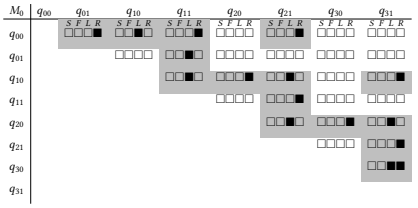


Fig. 7. Initial parse chart configuration.

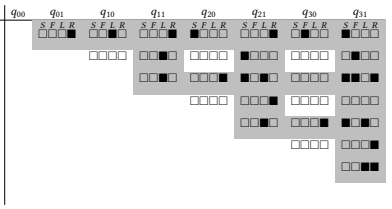


Fig. 8. Final parse chart configuration.

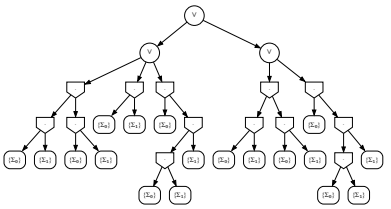


Fig. 9. Regular expression denoting $\mathcal{L}(G_\cap)$.

5 MEASURING THE LANGUAGE INTERSECTION

We will now attempt to put a probability distribution over the language intersection. We will start with a few cursory but illumative approaches, then proceed towards a more refined solution.

5.1 Mode collapse

Ordinarily, we would use top-down PCFG sampling, however in the case of non-recursive CFGs as the case for $G_\cap$, this method is highly degenerate, exhibiting poor sample diversity. Consider an illustrative pathological case for top-down ancestral (TDA) sampling:

$$S \to A\ B\ (0.9999) \qquad S \to C\ C\ (0.0001)$$

$$A \to a\ (1) \qquad B \to b\ (1) \qquad C \to a\ \left(\frac{1}{26}\right) \mid \ldots \mid z\ \left(\frac{1}{26}\right)$$

TDA sampling will almost always generate the string *ab*, but most of the language is concealed in the hidden branch, $S \to CC$. Although contrived example, it illustrates precisely why TDA sampling is unviable: we want a sampler that matches the true distribution over the finite CFL, not the PCFG's local approximation thereof.

5.2 Exact enumeration

To correct for mode collapse, a brute force solution would be to generate every path and rank it by probability. While this is unviable due its worst case complexity, it bears mentioning due to its global optimality. In certain cases, it can be realized when the intersection language is small.

To enumerate, we first need $|\mathcal{L}(e)|$, which is denoted $|e|$ for brevity.

Definition 5.1 (Cardinality). $$|e| : E \to \mathbb{N} = \begin{cases} 1 & \text{if } R \in \Sigma \\ x \times z & \text{if } e = x \cdot z \\ x + z & \text{if } e = x \vee z \end{cases}$$

Theorem 5.2 (Enumeration). *To enumerate, invoke* $\bigcup_{i=0}^{|R|} \{\mathit{enum}(R, i)\}$*:*

$$\mathit{enum}(e, n) : E \times \mathbb{N} \to \Sigma^* = \begin{cases} e & \text{if } R \in \Sigma \\ \mathit{enum}\left(x, \lfloor \frac{n}{|z|} \rfloor\right) \cdot \mathit{enum}\left(z,\ n \bmod |z|\right) & \text{if } e = x \cdot z \\ \mathit{enum}\left((x, z)_{\min(1, \lfloor \frac{n}{|x|} \rfloor)},\ n - |x| \min(1, \lfloor \frac{n}{|x|} \rfloor)\right) & \text{if } e = x \vee z \end{cases}$$

This can be converted to a uniform sampler by drawing integers without replacement using a pseudorandom number generator, however, if $|e|$ is very large this can fail to capture modes.

5.3 Ambiguity

Another approach would be to sample whole trees and rerank them by their PCFG score. More pernicious is the issue of ambiguity. Since the CFG can be ambiguous, this causes certain repairs to be overrepresented, resulting in a subtle bias. Consider for example,

Lemma 5.3. *If the FSA, α, is ambiguous, then the intersection grammar, $G_\cap$, can be ambiguous.*

Proof. Let ℓ be the language defined by $G = \{S \to LR, L \to (, R \to)\}$, where $\alpha = L(\underset{\sim}{\sigma}, 2)$, the broken string $\underset{\sim}{\sigma}$ is `)(`, and $\mathcal{L}(G_\cap) = \ell \cap \mathcal{L}(\alpha)$. Then, $\mathcal{L}(G_\cap)$ contains the following two identical repairs: `)()` with the parse $S \to q_{00}Lq_{21}\ q_{21}Rq_{22}$, and `()` with the parse $S \to q_{00}Lq_{11}\ q_{11}Rq_{22}$. □

We would like the underlying sample space to be a proper set, *not* a multiset.

6 IMPLEMENTATION

The implementation essentially consists of four stages, each dependent on its predecessor.

(1) `lev_build` : $\Sigma^{|Q|-1} \times \mathbb{N}^3 \to$ NFA – constructs a Levenshtein NFA from the broken string.
(2) `cfl_fixpt` : NFA $\times$ CFG $\to \mathbb{B}^{|Q|\times|Q|\times|V|}$ – computes the matrix exponential.
(3) `reg_build` : $\mathbb{B}^{|Q|\times|Q|\times|V|} \times$ CFG $\to$ REG – constructs the regular expression for $G_\cap$.
(4) `reg_dcode` : REG$\times\mathbb{N}^{|\Sigma|^{c\approx 3}}\times\mathbb{N} \to (\Sigma^+)^{k\approx 10}$ – returns a small set of the most probable repairs.

We will now explore the imperative pseudocode for each stage, starting with the Levenshtein automata constructor, which is a straightforward translation of the inference rules in Sec. 4.3.

Algorithm 1 `lev_build` pseudocode

procedure `lev_build`$(\sigma : \Sigma^n, d_{\max} : \mathbb{N})$ ▷ Takes a string and maximum edit distance.
  $Q, \delta \leftarrow \varnothing$
  for $\langle h, j, i, k\rangle$ **in** $[0, n]^2 \times [0, d_{\max}]^2$ **do**

$$\delta' \leftarrow \begin{cases} q_{h,i} \xrightarrow{[\neq\sigma_{j+1}]} q_{j,k} & \text{if } h = j \qquad \wedge\, i = k-1 \\ q_{h,i} \xrightarrow{[\neq\sigma_{j}]} q_{j,k} & \text{if } h = j-1 \qquad \wedge\, i = k-1 \\ q_{h,i} \xrightarrow{[=\sigma_{j}]} q_{j,k} & \text{if } h = j-1 \qquad \wedge\, i = k \\ q_{h,i} \xrightarrow{[=\sigma_{j}]} q_{j,k} & \text{if } 1 \le j-h-1 \le d_{\max} \wedge\, 1 \le k-i \le d_{\max} \end{cases}$$

  $Q \leftarrow Q \cup \{\pi_1(\delta'), \pi_3(\delta')\}, \delta \leftarrow \delta \cup \{\delta'\}$
  $I \leftarrow \{q_{0,0}\}, F \leftarrow \{q_{i,j} \mid n - i + j \le d_{\max}\}$
  return $\langle Q, \Sigma, \delta, I, F\rangle$ ▷ Returns a Levenshtein automaton.

Next, the chart parser expects an acyclic NFA, a CNF grammar and returns a Boolean 3-tensor.

Algorithm 2 `cfl_fixpt` pseudocode

Require: CFG must be in CNF and the NFA must be ε-free and acyclic (i.e, denote a finite langauge).
procedure `cfl_fixpt`$\big(\langle \Sigma, V, P, S\rangle : \text{CFG}, \langle Q, \Sigma, \delta, I, F\rangle : \text{NFA}\big)$
  $R : \mathbb{B}^{|Q|\times|Q|} \leftarrow \left[\blacksquare \textbf{ if } \exists \sigma \in \Sigma^+ \mid q \overset{\sigma}{\rightsquigarrow} q' \textbf{ else } \square\right]_{q,q':Q}$ ▷ Solve for reachability matrix.
  $M : \mathbb{B}^{|Q|\times|Q|\times|V|} \leftarrow \left[\blacksquare \textbf{ if } \exists v : V, w : \Sigma \mid (v \to w) \in P \wedge (q \xrightarrow{s} q') \in \delta \textbf{ else } \square\right]_{q,q':Q,\, v:V}$
  for i **in** $\left[0, \lceil \log_2(|Q|)\rceil\right]$ **do** ▷ Solves matrix exponential, $\exp(M_0)$.
    DONE $\leftarrow \blacksquare$
    for $\langle p, r, w\rangle$ **in** $Q^2 \times V$ **do** ▷ Iterates one step of $M_{i+1} = M_i + M_i^2$.
      if $M[p, r, w]$ **or not** $R[p, r]$ **then continue**
      $Q_{pr} \leftarrow \{q : Q \mid R[p, q] \wedge R[q, r]\}$ ▷ Consider reachable states between p and r.
      $M[p, r, w] \leftarrow \blacksquare$ **if** $\exists q : Q_{pr}, x, z : V \mid M[p, q, x] \wedge M[q, r, z] \wedge (w \to xz) \in P$ **else** $\square$
      if $M[p, r, w]$ **then** DONE $\leftarrow \square$
    if DONE **then break**
  return M ▷ Returns the completed Boolean parse chart.

Note we may short-circuit for three reasons, if: $M_{i+1} = M_i$, when two states q, q' are unreachable, or whenever a $\langle q, q', v\rangle$ is already present. Once we obtain M_∞, we can immediately tell whether $\ell_\cap \neq \varnothing$ by checking $M_\infty[q, q', S] = \blacksquare$ for some $q : I, q' : F$. Otherwise if no such q' exists, then $\ell_\cap$ must be empty and $d_{\max}$ should be enlarged before proceeding.

Now we can perform a second sweep over nonempty entries of the Boolean parse chart, reconstructing the provenance of each $\langle q, q', v\rangle$ constituent. For compactness it will be convenient to use a pointer-based representation of the regular expression instead of manipulating strings.

Algorithm 3 `reg_build` pseudocode

Require: $M_{\mathbb{B}}[q : I, q' : F, S] = \blacksquare$ for some q, q' and $M_{\mathbb{B}} = M_{\mathbb{B}} + M_{\mathbb{B}}^2$.

procedure `reg_build`$\big(M_{\mathbb{B}} : \mathbb{B}^{|Q|\times|Q|\times|V|}, \langle \Sigma, V, P, S\rangle : \text{CFG}, \langle Q, \Sigma, \delta, I, F\rangle : \text{NFA}\big)$

  $P : \mathbb{B}^{|Q|\times|Q|} \leftarrow \Big[\blacksquare \textbf{ if } \exists q : Q, v, v' : V \mid M_{\mathbb{B}}[p, q, v] \wedge M_{\mathbb{B}}[q, r, v'] \textbf{ else } \square\Big]_{p,r:Q}$

  $M : \text{REG}^{|Q|\times|Q|\times|V|} \leftarrow \Big[\{w \mid M[q, q', v] \wedge (q \xrightarrow{w} q') \in \delta \wedge (v \to w) \in P\}\Big]_{q,q':Q,v:V}$

  for i **in** $\big[0, \lceil \log_2(|Q|)\rceil\big]$ **do**

    $M' \leftarrow M$

    for $\langle p, r, w\rangle$ **in** $Q^2 \times V$ **do**

      if not $M_{\mathbb{B}}[p, r, w]$ **then continue**

      $Q_{pr} \leftarrow \{q : Q \mid P[p, q] \wedge P[q, r]\}$ ▷ Consider parseable states between p and r.

      $M'[p, r, w] \leftarrow M[p, r, w] \vee \bigvee_{\substack{q:Q_{pr}\\ x,z:V}} \big\{M[p, q, x] \cdot M[q, r, z] \mid (w \to xz) \in P\big\}$

    if $M = M'$ **then break else** $M \leftarrow M'$

  return $\bigvee_{q:I,q':F} M[q, q', S]$ ▷ Union regexes for all total parses yielding S.

Finally, once we have the expression for $G_\cap$, we can decode it to extract a small set of candidates. Various strategies are possible here, and we opt for the simplest one. We use two priority queues to store partial and total trajectories, which are ranked by probability as estimated by a pretrained c-gram count tensor, C. Partial trajectories are greedily extended until termination, after which the trajectory it is diverted to the total queue, and the top-k total trajectories are returned.

Algorithm 4 `reg_dcode` pseudocode

Require: We expect the shortest word to exceed the Markov order in length, $c < |\sigma|, \forall \sigma : \mathcal{L}(e)$.

procedure `reg_dcode`$\big(e : \text{REG}, C : \mathbb{N}^{|\Sigma|^{c\asymp 3}}, k : \mathbb{N}\big)$

  $\mathcal{T} \leftarrow [], \mathcal{E} \leftarrow \big[\langle \varepsilon^{c-1}, e \cdot \varepsilon^{c-1}, 0\rangle\big]$ ▷ Initialize total and partial trajectories.

  let $P(s : \Sigma \mid \sigma : \Sigma^{\geq c-1}) = \dfrac{C[\sigma_{|\sigma|-c+1,|\sigma|} \cdot s] + 1}{\sum_{s'\in\Sigma} C[\sigma_{|\sigma|-c+1,|\sigma|} \cdot s']}$ ▷ Define Markov transition probability.

  repeat

    $\langle \sigma, e, p\rangle \leftarrow$ **pop** $\mathcal{E}_0$ **off** $\mathcal{E}$

    $\mathcal{E}' \leftarrow \big[\langle \sigma \cdot a, \partial_a e, p + \ln P(a \mid \sigma)\rangle \mid a \in \texttt{follow}(e)\big]$

    $\mathcal{T} \leftarrow \mathcal{T} +\!\!+ \big[\langle \sigma, p\rangle \mid \langle \sigma, e, p\rangle \in \mathcal{E}' \wedge \varepsilon \in \mathcal{L}(e)\big]$

    $\mathcal{E} \leftarrow \big[\langle \sigma, e, p\rangle \in (\mathcal{E} +\!\!+ \mathcal{E}')$ **sorted by** $p\big]$

  until interrupted or $\mathcal{E}$ is empty.

  return $[\sigma \mid \langle \sigma, p\rangle \in \mathcal{T}_{0..k}$ **sorted by** $p]$ ▷ Skim off top-k repairs by probability.

Now, we have our shortlist of repairs and after cosmetic postprocessing, can present them to the user. With this approach, we can quickly generate a representative subset of $\ell_\cap$ within a fixed latency budget, e.g., 100ms, or otherwise terminate early should we succeed in exhaustively generating it.

6.1 GPU translation

Each stage is translatable to a GPU format, especially matrix multiplication which GPUs are specifically designed to accelerate. Our strategy will be to treat each stage as a massively parallel map-reduce job, where each thread is responsible for populating a single entry of a large buffer, independently and without communication.

We will make the simplifying assumption that each GPU kernel is a pure function that takes as input a coordinate triple $r, c, v : \mathbb{N}$ and one or more flat buffers $b_1 : \mathbb{N}^{d_1}, \ldots b_n : \mathbb{N}^{d_n}$ containing encoded data, does some arithmetic, and returns a single output buffer, $b_{\text{out}} : \mathbb{N}^d$. The main challenge of GPU programming becomes mapping complex datatypes to and from an integer format.

Morally, each $\langle r, c, v\rangle$ triple will dispatch a single independent thread which reads from the input buffers and has exclusive write access to a contiguous region of the output buffer. Absent a GPU, this can be rewritten as a triply-nested loop subject to latency overhead. On a GPU, threads are effectively executed simultaneously but memory must be sized ahead of time as no dynamic allocation is allowed during a GPU kernel's execution.

For CFG and NFA datatypes, we elect to use a dense representation $\mathbb{B}^{|V|\times|V|\times|V|}$ and $\mathbb{B}^{|Q|\times|Q|\times|\Sigma|}$ due to the tripartite coordinate structure and thread dispatching API. While these datatypes can be encoded sparsely as $\mathbb{N}^{3|P|}$ and $\mathbb{N}^{3|\delta|}$, for most repair instances and memory configurations representation size is not a bottleneck. It will be helpful to define two indices $\texttt{enc} : \Sigma \to 2^V$ and $\texttt{dec} : V \to 2^\Sigma$ for nonterminal encoding and decoding, and bijections $\Sigma \leftrightarrow \mathbb{N}$, $V \leftrightarrow \mathbb{N}$ for getting into and out of the integer domain – these we omit for brevity but are trivial to define.

The parse chart M can be represented as an integer matrix $\texttt{uint32}^{|Q|\times|Q|\times|V|}$, where the layout for each entry encodes up to three properties of each $\langle q, q', v\rangle$ triple – (1) equality or disequality of the nominal predicate (2) the terminal in a unit production and (3) whether or not the nonterminal v can parse a path from q to q'. This datatype can be reduced to `uint16` or `uint8` if $|\Sigma| \ll 2^{30}$ or if GPU memory is a bottleneck. This is depicted below in little-endian format.

$$\Big[\overbrace{\blacksquare}^{=/\neq}, \overbrace{\square, \square, \ldots, \blacksquare, \square, \blacksquare}^{s:\Sigma\Leftrightarrow\mathbb{B}^{30}}, \overbrace{\blacksquare}^{v:V}\Big] : \texttt{uint32}$$

Once `cfl_fixpt` is complete, we can calculate the total amount of memory needed to allocate $G_\cap$ by counting constituents in the parse chart. The GRE datatype is slightly more complex to flatten, as being an algebraic datatype it can be simplified in various ways. We will use the following memory layout to encode a GRE, however this choice is somewhat arbitrary,

$$\Big[\overbrace{\underbrace{2}_{\texttt{bp}_0}, \underbrace{7}_{\texttt{bp}_1}, \ldots, \underbrace{1}_{\texttt{bp}_{c-2}}, \underbrace{3}_{\texttt{bp}_{c-1}}}^{\texttt{bp\_counts}}, \overbrace{0, 4, \ldots, \underbrace{n-8}_{\texttt{bp}_{c-2}}, \underbrace{n-6}_{\texttt{bp}_{c-1}}}^{\texttt{bp\_offsets}}, \overbrace{\underbrace{59, 83, 64, 152}_{\texttt{bp}_0}, \ldots, \underbrace{34, 83}_{\texttt{bp}_{c-2}}, \underbrace{22, 74, 74, 90, 16, 66}_{\texttt{bp}_{c-1}}}^{\texttt{bp\_storage}}\Big] : \texttt{uint32}^n$$

where each $\texttt{bp}_i$ represents a nonempty $\langle q, q', v\rangle$ constituent with at least one back-pointer pair, $\texttt{bp\_count}(p, r, w) = \left|\left\{\langle q, x, z\rangle \mid M[p, q, x] \wedge M[q, r, z] \wedge (w \to xz) \in P\right\}\right|$ counts the number of unique backpointers held by each nonterminal w parseable from $p \rightsquigarrow r$, and `bp_storage` stores the pointers to memory locations in the same data structure. These pointers must also be tied to locations in the parse chart to obtain the terminal subsets for adjacent states and unit nonterminals when decoding.

7 EVALUATION

We call our method Tidyparse and consider the following research questions:

- **RQ 1**: What statistical properties do human repairs exhibit? (e.g., length, edit distance)
- **RQ 2**: How performant is Tidyparse at fixing syntax errors? (i.e., vs. Seq2Parse and BIFI)
- **RQ 3**: Which design choices are most significant? (e.g., sampling, decoding, parallelism)

We address **RQ 1** in § 7.2 by analyzing the distribution of natural code snippet lengths and edit distances, **RQ 2** in § 7.3 by comparing Tidyparse against two existing syntax repair baselines, and **RQ 3** in § 7.4 by ablating various design choices and evaluating the impact on repair precision.

7.1 Experimental setup

We use syntax errors and fixes from the Python language to validate our approach. Python source code fragments are abstracted as a sequence of lexical tokens using the official Python lexer, erasing numbers and identifiers, but retaining all other keywords. Accuracy is evaluated across a test set by checking for lexical equivalence with the ground-truth repair, following Sakkas et al. (2022) [42].

To evaluate accuracy, we use the Precision@k statistic, which measures the frequency of repairs in the top-k results matching the true repair. Specifically, given a repair model, $R : \Sigma^* \to 2^{\Sigma^*}$ and a test set $\mathcal{D}_{\text{test}}$ of pairwise aligned errors ($\sigma^\dagger$) and fixes (σ'), we define Precision@k as:

$$\text{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{\langle \sigma^\dagger, \sigma' \rangle \in \mathcal{D}_{\text{test}}} \mathbb{1}\left[\sigma' \in \underset{\boldsymbol{\sigma} \subseteq R(\sigma^\dagger), |\boldsymbol{\sigma}| \leq k}{\text{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} \text{Score}(\sigma)\right] \tag{9}$$

This is a variation on a standard metric used in information retrieval and a common way of measuring the quality of ranked results in machine translation and recommender systems. Precision@All or completeness may be seen as a special case where $k = \infty$.

By default, Tidyparse uses the DFA decoder (Alg. 3) for all experiments, however, we also include a comparison with a naïve rejection-based edit sampler, as well as enumerative sampling with PCFG reranking (Alg. 1) and n-gram reranking (Alg. 2) in our ablation study (§ 7.4).

We compare our method against two external baselines, Seq2Parse and Break-It-Fix-It (BIFI) [52] on a single test set. This dataset [51] consists of 20k naturally-occurring pairs of Python errors and their corresponding human fixes from StackOverflow, and is used to compare the precision of each method at blind recovery of the ground truth repair across varying edit distances, snippet lengths and latency cutoffs. We preprocess all source code by filtering for broken-fixed snippet pairs shorter than 100 tokens and fewer than five Levenshtein edits apart, whose broken and fixed form is rejected and accepted, respectively, by the Python 3.8.11 parser. We then balance the dataset by sampling an equal number of repairs from each length and Levenshtein edit distance.

The Seq2Parse and BIFI experiments were conducted on a single Nvidia V100 GPU with 32 GB of RAM. For Seq2Parse, we use the default pretrained model provided in commit `7ae0681` [4]. Since it was unclear how to extract multiple repairs from their model, we only take a single repair prediction. For BIFI, we use the Round 2 breaker and fixer from commit `ee2a68c`[5], the highest-performing model reported by the authors, with a variable-width beam search to control the number of predictions, and let the BIFI fixer model predict the top-k repairs, for $k = \{1, 5, 10, 2 \times 10^4\}$.

The language intersection experiments were conducted on a MacBook M4 Max with 16GB of memory. To train our scoring function, we use an order-4 Markov chain trained on 55 million BIFI tokens. Training takes 10 minutes, after which re-ranking is nearly instantaneous. Sequences are scored using NLL with Laplace smoothing and our evaluation measures Precision@{1, 5, 10, All}.

[4] https://github.com/gsakkas/seq2parse/tree/7ae0681f1139cb873868727f035c1b7a369c3eb9
[5] https://github.com/michiyasunaga/BIFI/tree/ee2a68cff8dbe88d2a2b2b5feabc7311d5f8338b

7.2 Dataset statistics

In the following experiments, we use a dataset of Python snippets consisting of 20,500 pairwise-aligned human errors and fixes from StackOverflow [51]. We preprocess the dataset to lexicalize all code snippets, then filter by length and distance shorter than 100 lexical tokens and under five edits, i.e., with Levenshtein distance under five lexical edits ($|\Sigma| = 50, |\sigma| < 100, \Delta(\sigma, \sigma') < 5$). We depict the length, edit distance, normalized edit locations and stability profile in Fig. 10.

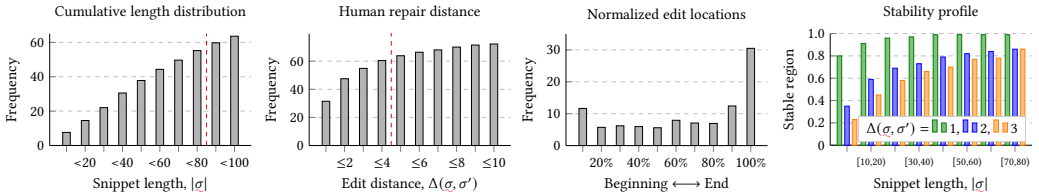


Fig. 10. Repair statistics across the StackOverflow dataset, of which Tidyparse can handle about half in under ~30s and ~150 GB. Larger repairs and edit distances are possible, albeit requiring additional time and memory.

We observe that slightly over half of the code snippet pairs in the StackOverflow dataset contain fewer than 100 tokens and five lexical edits, which our method can easily handle (§ 7.3). The distribution across edit locations indicates a large fraction of human edits occur near the boundaries of the broken code snippet, however we do not exploit this prior anywhere in the repair process.

For the stability profile, we enumerate repairs for each syntax error and estimate the average fraction of all edit locations that were never altered by any repair in the $L(\sigma, \Delta(\sigma, \sigma'))$-ball. For example, on average roughly half of the string is stable for 3-edit syntax repairs in the $[10 - 20)$ token range, whereas 1-edit repairs of the same length could modify only $\sim 10\%$ of all locations. For a fixed edit distance, we observe an overall decrease in the number of degrees of caret freedom with increasing length, which intuitively makes sense, as the repairs are more heavily constrained by the surrounding context and their locations grow more concentrated relative to the entire string.

7.3 StackOverflow evaluation

For our first experiment, we measure the precision of our repair procedure at various lengths and Levenshtein distances. We rebalance the StackOverflow dataset across each length interval and edit distance, sample uniformly from each category and compare Precision@1 of our method against Seq2Parse, vanilla BIFI and BIFI with a beam size and precision at 2×10^4 distinct samples.

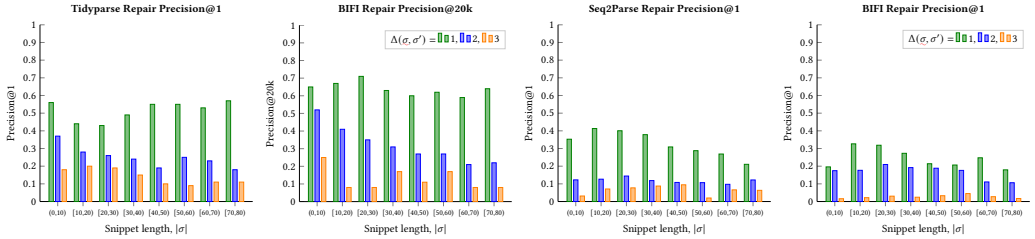


Fig. 11. Tidyparse, Seq2Parse and BIFI repair precision at various lengths and Levenshtein distances.

As we can see, Tidyparse has a highly competitive top-1 precision versus Seq2Parse and BIFI across all lengths and edit distances, and attains a significant advantage in the few-edit regime. The Precision@1 of our method is even competitive with BIFI's Precision@20k, whereas our Precision@All is Pareto-dominant across all lengths and edit distances, while requiring only a fraction of the data and compute. We report the raw data from these experiments in Appendix D.

Next, we measure the precision at various ranking cutoffs and wall-clock timeouts. Our method attains the same precision as Seq2Parse and BIFI for 1-edit repairs at comparable latency, however Tidyparse takes longer to attain the same precision for 2- and 3-edit repairs. BIFI and Seq2Parse both have subsecond single-shot latency but are neural models trained on a much larger dataset.

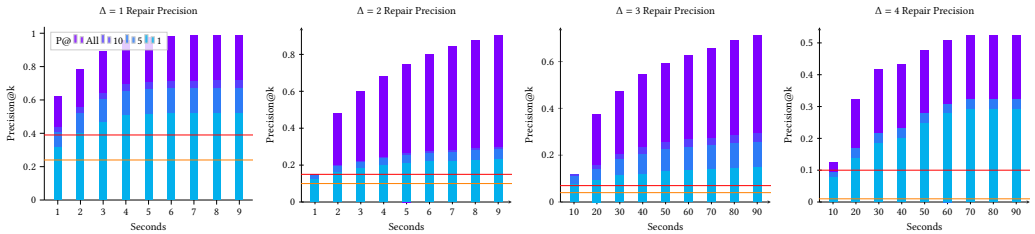


Fig. 12. Human repair benchmarks. Note the y-axis across different edit distance plots has varying ranges. The red line indicates Seq2Parse and the orange line indicates BIFI's Precision@1 on the same repairs.

We present a Sankey diagram of our repair pipeline in Fig. 13. We drew 2247 total repairs from the StackOverflow dataset balanced evenly across lengths and edit distances ($\lfloor |\sigma|/10 \rfloor \in [0, 8]$, $\Delta(\sigma, \sigma') < 4$) with a timeout of 30s and tracked individual outcomes. In 101 cases, the intersection grammar was too large to construct and threw an out-of-memory (OOM) error, in 45 cases the human repair was not recognized, in 253 cases the sampler timed out before drawing the human repair, in 1226 cases the human repair was drawn but not ranked first, and in the remaining 622 cases the first prediction matched the human repair.

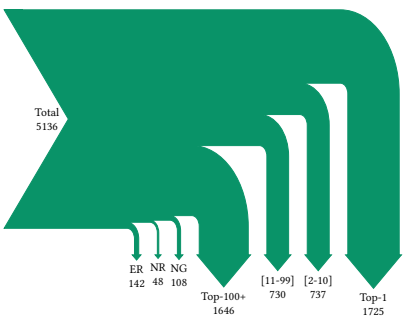


Fig. 13. Outcomes in the repair pipeline.

The remaining experiments in this section were run on a 10-core ARM64 M1 with 16 GB of memory. We balance the StackOverflow dataset across Levenshtein distances, then measure the number of samples required to draw the exact human repair across varying Levenshtein radii. This tells us of how many samples are required on average to saturate the admissible set.

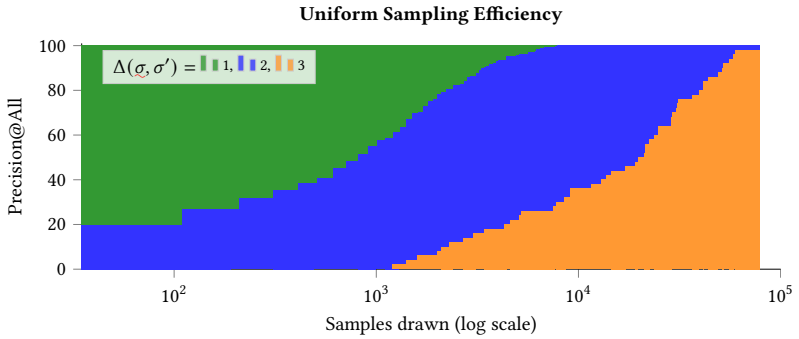


Fig. 14. Sample efficiency of Tidyparse at varying Levenshtein radii. After drawing up to $\sim 10^5$ samples without replacement we can usually retrieve the human repair for almost all repairs fewer than four edits.

End-to-end throughput varies significantly with the edit distance of the repair. Some errors are trivial to fix, while others require a large number of edits to be sampled before the ground truth is discovered. We evaluate throughput by sampling patches across invalid strings $|\sigma| \leq 40$ from the StackOverflow dataset balanced across length and distance, and measure the total number of unique valid patches discovered, as a function of string length and edit distance $\Delta \in [1, 4]$. Each trial is terminated after 10 seconds, and the experiment is repeated across 7.3k total repairs. Note the y-axis is log-scaled, as the number of admissible repairs increases sharply with edit distance. Our approach discovers a large number of syntactic repairs in a relatively short amount of time, and is able to quickly saturate the admissible set for $\Delta(\sigma, \sigma') \in [1, 4]$ before timeout.

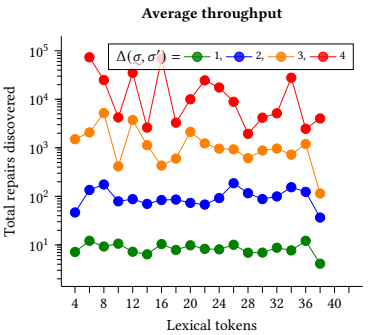


Fig. 15. Distinct repairs found in 30s.

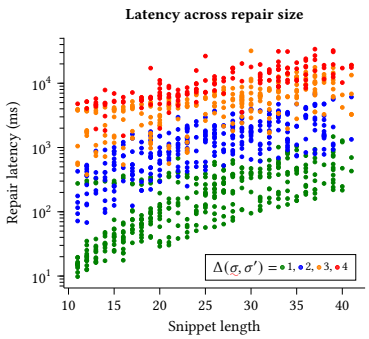


Fig. 16. End-to-end repair timings.

In Fig. 16, we plot the end-to-end repair timings by collecting 1000 samples balanced across length and edit distance, then measure the wallclock time until the sampler retrieves the human repair and report the log latency. While short repairs finish quickly, latency is positively correlated with length and edit distance. Our method is typically able to saturate the admissible set for 1- and 2-edit repairs before timeout, while 4+-edit throughput starts becoming constrained by compute around 30s, when Python's admissible set approaches a volume of 10^5 valid edits. This bottleneck can be relaxed with a longer timeout or additional CPU cores. We anticipate that a much longer delay will begin to tax the patience of most users, and so we consider 30s a reasonable upper bound for repair latency. As we will now show, end-to-end latency can be improved by doing rejection sampling, albeit at the cost of naturalness and sample efficiency.

7.4 Subcomponent ablation

Originally, we used an adaptive rejection-based sampler, which did not sample directly from the admissible set, but the entire Levenshtein ball, and then rejected invalid samples. Although rejection sampling has a much lower minimum latency threshold to return admissible repairs, i.e., a few seconds at most, the average time required to attain a desired precision on human repairs is much higher. We present the results from the rejection-based evaluation for comparison below.

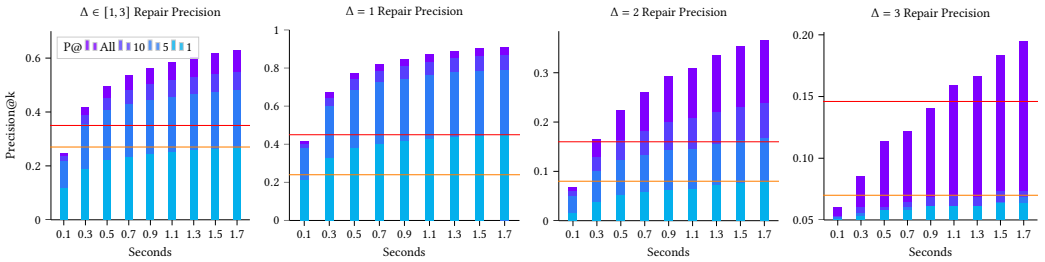
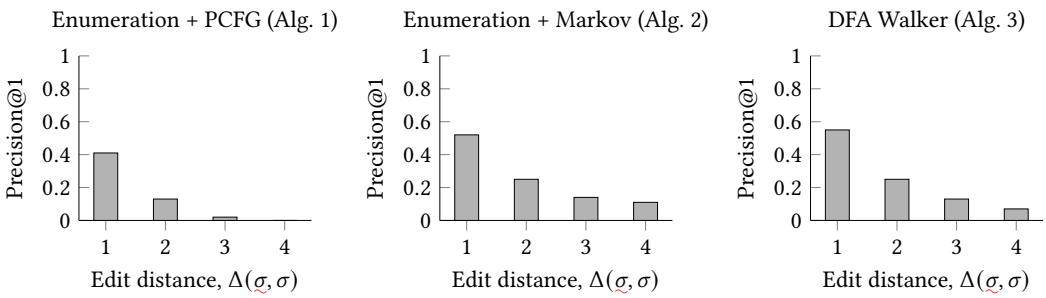


Fig. 17. Adaptive sampling repairs. The red line indicates Seq2Parse Precision@1, and the orange indicates BIFI's precision at single-shot repair, all three of which were evaluated on the exact same repairs.

We also evaluate Seq2Parse on the same dataset. Seq2Parse only supports Precision@1 repairs, and so we only report Seq2parse Precision@1 from the StackOverflow benchmark for comparison. Unlike our approach, which only produces syntactically correct repairs, Seq2Parse and BIFI also produce syntactically incorrect repairs in practice. The overall latency of Seq2Parse varies depending on the length of the repair, averaging 1.5s for $\Delta = 1$ to 2.7s for $\Delta = 3$, across the entire StackOverflow dataset, while BIFI consistently achieves subsecond latency across all repairs and distances.

Next, we conduct an ablation study across three decoding strategies to compare their relative effectiveness. In each experiment, we balance the StackOverflow dataset across edit distances and run the candidate sampler for up to 30 seconds. In Alg. **??**, we use the enumerative sampler and rank all repairs by PCFG score, in Alg. **??**, we use the same approach but rank the repairs by n-gram log-likelihood, and in Alg. **??**, we translate the BH intersection grammar into a DFA then sample trajectories according to a n-gram transition probability, as described in § **??**. We compare the Precision@1 of each method at recovering the ground truth human repair.



In general, n-gram likelihood appears to have a significant advantage over PCFG scoring, however this margin may decrease with PCFG models that consider higher-order nonterminal dependencies. Alg. **??** is efficient, but also the least precise, being a poor model for lexical alignment. Alg. **??** offers competitive precision for Python, but can produce duplicate samples in highly ambiguous

CFGs. Alg. ?? has the best performance across all edit distances and languages, but is also the most computationally expensive, requiring a determinization and minimization preprocessing step.

Finally, we evaluate the impact of increased parallelism on repair throughput. We balance the StackOverflow dataset across edit distances and run DFA sampler for up to 30 seconds, then measure the total number of unique valid repairs discovered as a function of the number of additional CPU cores assigned, which we exercise to both construct the intersection grammar and sample from it.

We measure the relative improvement in throughput (measured by the number of distinct repairs found after 30s) as a function of the number of additional CPU cores, averaged across 1000 trials. We observe from Fig.18 the relative throughput increases logarithmically with the number of additional CPU cores, with at least four CPU cores needed to offset the parallelization overhead. Generally, increasing parallelism only helps when the size of the admissible set is large enough to absorb the additional computation, which is seldom the case for small-radii Levenshtein balls. Further speedups are likely possible to realize by rewriting the sampler in CUDA, an engineering challenge which we leave for future work.

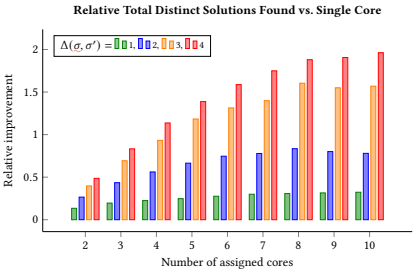


Fig. 18. Observed improvement in throughput relative to total CPU cores assigned.

8 DISCUSSION

The main lesson we draw from our experiments is that it is possible to leverage compute to compete with large language models on practical program repair tasks. Though sample-efficient, their size comes at the cost of expensive training, and domain adaptation requires fine-tuning or retraining on pairwise repairs. Our approach uses a small grammar and a relatively cheap ranking metric to achieve significantly higher precision. This allows us to repair errors in languages with little to no training data and provides far more flexibility and controllability during the repair process.

Our primary insight leading to state-of-the-art precision is that repairs are typically concentrated near the center of a small Levenshtein ball, and by enumerating or sampling it carefully, then reranking repairs by naturalness, one can achieve significantly higher precision than one-shot neural repair. This is especially true for small-radii Levenshtein balls, where the admissible set is small enough to be completely enumerated and ranked. For larger radii, we can still achieve competitive precision by using an efficient decoder to sample the admissible set.

There is a clear tradeoff between latency and precision for any repair model. While existing neural syntax repair models scale poorly with additional time, Tidyparse is highly effective at exchanging more time for higher precision. We find that the Precision@1 of our method is competitive with BIFI's Precision@20k, while requiring only a fraction of the data and compute for training and inference. As Tidyparse uses its own grammar, it can sample directly from the formal language specification and does not require a stochastic language model to suggest nearby valid repairs, only to rank them by naturalness. The emphasis on completeness is especially useful for discovering small or contextually unlikely repairs, which may be overlooked by neural models.

Although latency and precision are ultimately the deciding usability factors, repair throughput is a crucial intermediate factor to consider when evaluating the performance of a repair system. Even with a perfectly accurate scoring function, if the correct repair is never retrieved, it will be for naught. By maximizing the total number of unique valid repairs, we increase the likelihood of retrieving natural repairs to give the scoring function the best chance of ranking them successfully. For this reason, we prioritize throughput heavily in our design (Def. ??) and evaluation (Fig. 15).

8.1 Limitations and future work

8.1.1 Naturalness. Firstly, Tidyparse does not currently support intersections between weighted CFGs and weighted finite automata, a la Pasti et al. [40]. This feature would allow us to put transition probabilities on the Levenshtein automaton corresponding to edit likelihood then construct a weighted intersection grammar. With this, one could preemptively discard unlikely productions from $G_\cap$ to reduce the complexity in exchange for relaxed completeness. We also hope to explore more incremental sampling strategies such as sequential Monte-Carlo [35].

The scoring function is currently computed over lexical tokens. We expect that a more precise scoring function could be constructed by splicing candidate repairs back into the original source code and then scoring plaintext, however this would require special handling for insertions and substitutions of names, numbers and identifiers that were absent from the original source code. For this reason, we currently perform the scoring in lexical space, which discards a useful signal, but even this coarse approximation is sufficient to achieve state-of-the-art precision.

Furthermore, the scoring function only considers each candidate repair $P_\theta(\sigma')$ in isolation, returning the most plausible candidate independent of the original error. One way to improve this would be to incorporate the broken sequence ($\underset{\sim}{\sigma}$), parser error message (m), original source (s), and possibly other contextual priors to inform the scoring function. This would require a more expressive probabilistic language model to faithfully model the joint distribution $P_\theta(\sigma' \mid \underset{\sim}{\sigma}, m, s, \ldots)$, but would significantly improve the precision of the generated repairs.

8.1.2 Complexity. Latency can vary depending on several factors including string length, grammar size, and critically the Levenshtein edit distance. This can be an advantage because, without any contextual or statistical information, syntax and minimal Levenshtein edits are often sufficiently constrained to identify a small number of valid repairs. It is also a limitation because the admissible set expands rapidly with edit distance and the Levenshtein metric diminishes in usefulness without a very precise metric to discriminate natural solutions in the cosmos of equidistant repairs.

Space complexity increases sharply with edit distance and to a lesser extent with length. This can be partly alleviated with more precise criteria to avoid creating superfluous productions, but the memory overhead is still considerable. Memory pressure can be attributed to engineering factors such as the grammar encoding, but is also an inherent challenge of language intersection. Therefore, managing the size of the intersection grammar by preprocessing the syntax and automaton, then eliminating unnecessary synthetic productions is a critical factor in scaling up our technique.

8.1.3 Toolchain integration. Lastly and perhaps most significantly, Tidyparse does not incorporate semantic constraints, so its repairs whilst syntactically admissible, are not guaranteed to be type safe. It may be possible to add a type-based semantic refinement to our language intersection, however this would require a more expressive grammatical formalism than CFGs naturally provide.

Program slicing is an important preprocessing consideration that has so far gone unmentioned. The current implementation expects pre-sliced code fragments, however in a more practical scenario, it would be necessary to leverage editor information to identify the boundaries of the repairable fragment. This could be achieved by analyzing historical editor states or via ad hoc slicing techniques.

Additionally, the generated repairs must be spliced back into the surrounding context, which requires careful editor integration. One approach would be to filter all repairs through an incremental compiler or linter, however, the latency necessary to check every repair may be non-negligible.

We envision a few primary use cases for Tidyparse: (1) helping novice programmers become more quickly familiar with a new programming language, (2) autocorrecting common typos among proficient but forgetful programmers, (3) as a prototyping tool for PL designers and educators, and (4) as a pluggable library or service for parser-generators and language servers.

9 RELATED WORK

Three important questions arise when repairing syntax errors: (1) is the program broken in the first place? (2) if so, where are the errors located? (3) how should those locations then be altered? Those questions are addressed by three theoretical areas, (1) parsing, (2) language equations and (3) syntax repair. We survey each of those areas, then turn our attention to more engineering-oriented research, including (4) string solving, (5) error-correction, (6) decoding and finally (7) neural program repair.

9.1 Parsing

Context-free language (CFL) parsing is the well-studied problem of how to turn a string into a unique tree, with many different algorithms and implementations (e.g., shift-reduce, recursive-descent, LR). Many of those algorithms expect grammars to be expressed in a certain form (e.g., left- or right- recursive) or are optimized for a narrow class of grammars (e.g., regular, linear).

General CFL parsing allows ambiguity (non-unique trees) and can be formulated as a dynamic programming problem, as shown by Cocke-Younger-Kasami (CYK) [41], Earley [21] and others. These parsers have roughly cubic complexity with respect to the length of the input string.

As shown by Valiant [49], Lee [33] and others, general CFL recognition is in some sense equivalent to binary matrix multiplication, another well-studied combinatorial problem with broad applications, known to be at worst subcubic. This reduction opens the door to a range of complexity-theoretic speedups to CFL recognition, however large constants tend to limit their practical utility.

From a more applied perspective, parsers are ubiquitous in present-day software engineering, but none are designed to handle arbitrary CFGs or recover from arbitrary errors. Parr and Quong introduce ANTLR [39] which can handle LL(k) grammars and offers an IDE plugin with limited support for error recovery. Scott and Johnstone [45] introduce GLL parsing, which supports linear-time parsing for LL grammars and cubic for arbitrary CFGs, but does not support error correction. Inspired by their work, we introduce a method for repairing small syntax errors in arbitrary CFLs.

9.2 Language equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [23] for the ALGOL language, language equations are essentially systems of inequalities with variables representing *holes*, i.e., unknown values, in the language or grammar. Solutions to these equations can be obtained using various fixpoint techniques, yielding members of the language. This insight reveals the true algebraic nature of CFLs and their cousins.

Being an algebraic formalism, language equations naturally give rise to a kind of calculus, vaguely reminiscent of Leibniz' and Newton's. First studied by Brzozowski [11, 12] and Antimirov [4], one can take the derivative of a language equation, which can be interpreted as a kind of continuation or language quotient, revealing the suffixes that complete a given prefix. This technique leads to an elegant family of algorithms for incremental parsing [1, 37] and automata minimization [10].

Bar-Hillel [5] establishes the closure of CFLs under intersection with regular languages, but does not elaborate on how to construct the corresponding grammar in order to recognize it. Beigel [8] and Pasti et al. [40] provide helpful insights into the construction of the intersection grammar, and Nederhof and Satta [38] specifically consider finite CFL intersections, but neither considers Levenshtein intersections. Our work specializes Bar-Hillel intersections to Levenshtein automata in particular, and more generally acyclic automata using a refinement of Salomaa's construction [43].

More concretely, we restrict our attention to language equations over CFLs whose variables coincide with edit locations in the source code of a computer program, and solutions correspond to syntax repairs. While prior work has studied the use of language equations for parsing [37], to our knowledge they were never specifically applied to code completion or syntax error correction.

9.3 Syntax repair

In finite languages, syntax repair corresponds to spelling correction, a more restrictive and largely solved problem. Schulz and Stoyan [44] construct a finite automaton that returns the nearest dictionary entry by Levenshtein edit distance. Though considerably simpler than syntax correction, their work shares similar challenges and offers insights for handling more general repair scenarios.

When a sentence is grammatically invalid, parsing grows more challenging. Like spelling, the problem is to find the minimum number of edits required to transform an arbitrary string into a syntactically valid one, where validity is defined as containment in a (typically) context-free language. Early work, including Irons [31] and Aho [2] propose a dynamic programming algorithm to compute the minimum number of edits required to fix an invalid string. Prior work on error correcting parsing only considers the nearest edit(s), and does not study edits of varying distance in the Levenshtein ball. Furthermore, the problem of repair is not generally well-posed, as there can be many valid solutions. We instead focus on maximum likelihood Levenshtein-CFL reachability, which attempts to find the most natural repair within a fixed Levenshtein distance.

9.4 String solving

There is related work on string constraints in the constraint programing literature, featuring solvers like CFGAnalyzer and HAMPI [32], which consider bounded context free grammars and intersections thereof. Bojańczyk et al. (2014) [9] introduce the theory of nominal automata. Around the same time, D'Antoni et al. (2014) introduce *symbolic automata* [17], a generalization of finite automata which allow infinite alphabets and symbolic expressions over them. Hague et al. (2024) [27] use Parikh's theorem in the context of symbolic automata to speed up string constraint solving, from which we draw partial inspiration for the Levenshtein-Bar-Hillel construction in § **??**. In none of the constraint programming literature we surveyed do any of the approaches specifically consider the problem of syntax error correction, which is the main focus of our work.

9.5 Error correcting codes

Our work focuses on errors arising from human factors in computer programming, in particular *syntax error correction*, which is the problem of fixing partially corrupted programs. Modern research on error correction, however, can be traced back to the early days of coding theory when researchers designed *error-correcting codes* (ECCs) to denoise transmission errors induced by external interference, e.g., collision with a high-energy proton, manipulation by an adversary or even typographical mistake. In this context, *code* can be any logical representation for communicating information between two parties (such as a human and a computer), and an ECC is a carefully-designed scheme which ensures that even if some portion of the message should become corrupted, one can still recover the original message by solving a linear system of equations. When designing ECCs, one typically assumes a noise model over a certain sample space, such as the Hamming [18, 48] or Levenshtein [6, 7, 34] balls, from which we draw inspiration for this work.

9.6 Decoding

Decoding is a key problem in machine translation, speech recognition, and other sequence-to-sequence tasks. Given a compressed encoding of some finite distribution, the goal is find the maximum likelihood samples. A classic example is Viterbi decoding, which is used to find the most likely sequence of transitions in a hidden Markov model (HMM) and is closely related to probabilistic parsing. For PCFGs, the problem is more challenging, as the solution space can be exponentially larger than HMMs relative to the number of transitions.

In particular, we care about the problem of *top-k decoding*, which attempts to find the exact or approximate k-most likely samples in order of decreasing likelihood. This is closely related to the k-best enumeration [22] problem, a carefully studied problem in graph theory and combinatorial optimization. An exact solution to this problem for large acyclic PCFGs is often intractable, but we can approximate it using a beam search or cube-pruning technique.

A popular solution to k-best decoding in the NLP literature is a technique called cube-pruning [13, 29, 30], which samples maximum likelihood paths through a hypergraph. We take inspiration from this technique, and adapt it to the setting of constrained decoding from finite CFGs. Our approach is also complementary to work by Zhang and McDonald [53], but specialized to language intersections.

An alternate line of work originates from combinatorics [25, 28] and Boltzmann sampling [20], which constructs a generating function for the language and samples it uniformly. Unlike our method, distinctness or convergence guarantees for arbitrary finite CFLs are not provided.

Another approach would be to use MCMC or sequential Monte Carlo (SMC) to steer a transformer-based LLM, as proposed by Lew et al. [35]. This technique shows promise for constrained sampling from LLMs, and could be adapted to improve sample efficiency. The downside is that distinctly sampling an LLM is unclear how to do properly, being a fundamentally non-Markovian process. One solution proposed by Shi and Bieber [46] assumes trace injectivity and constructs a trie, however their solution is not stateless and can introduce a significant latency overhead.

Our approach is complementary to existing work in constrained decoding. The bijection proposed in Eq. **??** guarantees that all repairs are well-formed and converge linearly to the exact top-k maximum likelihood samples. This method is completely stateless and can be used to enumerate a bounded Levenshtein ball with linear parallelization speedup. Alternately, in the case of approximate ranked repair over a very large sample space, this technique can be adapted to sample with high probability a representative subset of the most likely sentences in a finite but large PCFG.

9.7 Neural program repair

More recently, probabilistic repair techniques have been introduced using neural models to predict the most likely correction [3, 15, 19]. These approaches typically employ large language models (LLMs) and treat the problem as a sequence-to-sequence transformation. While capable of generating natural repairs, these models are susceptible to misgeneralization, costly to train, and challenging to customize thereafter. Furthermore, the generated repairs are not necessarily sound without additional filtering, and we observe the released models often hallucinate false positive repairs.

In particular, two papers stand out being closely related to our own: Break-It-Fix-It (BIFI) [52] and Seq2Parse [42]. BIFI adapts techniques from semi-supervised learning to generate synthetic errors in clean code and fixes them. This reduces the need for pairwise training data, but tends to generalize poorly to lengthy or out-of-distribution repairs. Seq2Parse combines a transformer-based model with an augmented version of the Early parser to suggest error rules, but only suggests a single repair. Our work differs from both in that we suggest multiple repairs at much higher precision, do not require a pairwise repair dataset, and can fix syntax errors in any language with a well-defined grammar. We note our approach is complementary to existing work in neural program repair, and may be used to generate synthetic repairs for training or employ an LLM for ranking.

Recent work by Merrill et al. [36] and Chiang et al. [14] suggest that the issue with generalization may be more foundational: transformer-based language models, a popular class of neural language models used in probabilistic program repair, are fundamentally less expressive than context-free grammars, which formally describe the syntax of most programming languages. This suggests such models, despite their useful approximation properties, are ill-suited for the task of end-to-end syntax repair. Yet, they may still be useful for resolving ambiguity between valid repairs of differing likelihood or searching a large sample space for the most likely repair.

10 CONCLUSION

Our work, while a case study on syntax repair, is part of a broader line of inquiry in program synthesis that investigates how to weave formal language theory and machine learning into helpful programming tools for everyday developers. In some ways, syntax repair serves as a test bench for integrating learning and language theory, as it lacks the intricacies of type-checking and semantic analysis, but is still rich enough to be an interesting challenge. By starting with syntax repair, we hope to lay the foundation for more organic hybrid approaches to program synthesis.

Two high-level codesign patterns have emerged to combine the naturalness of neural language models with the precision of formal methods. One seeks to filter the outputs of a generative language model to satisfy a formal specification, typically by some form of rejection sampling. Alternatively, some attempt to use language models to steer an incremental search for valid programs via a reinforcement learning or hybrid neurosymbolic approach. However, implementing these strategies is often painstaking and their generalization behavior can be difficult to analyze.

In our work, we take a more pragmatic tack - by incorporating the distance metric into a formal language, we attempt to exhaustively enumerate repairs by increasing distance, then use the stochastic language model to sort the resulting solutions by naturalness. The more constraints we can incorporate into formal language, the more efficient sampling becomes, and the more precise control we have over the output. This reduces the need for training a large, expensive language model to relearn syntax, and allows us to leverage compute for more efficient search and ranking.

There is a delicate balance in formal methods between soundness and completeness. Often these two seem at odds because the target language is too expressive to achieve them both simultaneously. In syntax repair, we also care about *naturalness*. Fortunately, syntax repair is tractable enough to achieve all three by modeling the problem using language intersection. Completeness helps us to avoid missing simple repairs that might be easily overlooked, soundness guarantees all repairs will be valid, and naturalness ensures the most likely repairs receive the highest priority.

From a usability standpoint, syntax repair tools should be as user-friendly and widely accessible as autocorrection tools in word processors. We argue it is possible to reduce disruption from manual syntax repair and improve the efficiency of working programmers by driving down the latency needed to synthesize an acceptable repair. In contrast with program synthesizers that require intermediate editor states to be well-formed, our synthesizer does not impose any constraints on the code itself being written and is possible to use in an interactive programming setting.

We have implemented our approach and demonstrated its viability as a tool for syntax assistance in real-world programming languages. Tidyparse is capable of generating repairs for invalid source code in a range of practical languages with little to no data required. We plan to continue expanding the prototype's autocorrection functionality to cover a broader range of languages and hope to conduct a more thorough user study to validate its effectiveness in practical programming scenarios.

DATA-AVAILABILITY STATEMENT

An artifact for Tidyparse is currently available as a browser application. [6]. The data and source code for the experiments contained in this paper will be made available upon publication.

[6] https://tidyparse.github.io

REFERENCES

[1] Michael D Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the complexity and performance of parsing with derivatives. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. 224–236.

[2] Alfred V Aho and Thomas G Peterson. 1972. A minimum distance error-correcting parser for context-free languages. SIAM J. Comput. 1, 4 (1972), 305–312.

[3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. Advances in Neural Information Processing Systems 34 (2021), 27865–27876. https://arxiv.org/pdf/2105.12787.pdf

[4] Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science 155, 2 (1996), 291–319.

[5] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung 14 (1961), 143–172.

[6] Daniella Bar-Lev, Tuvi Etzion, and Eitan Yaakobi. 2021. On Levenshtein Balls with Radius One. In 2021 IEEE International Symposium on Information Theory (ISIT). 1979–1984. https://doi.org/10.1109/ISIT45174.2021.9517922

[7] Leonor Becerra-Bonache, Colin de La Higuera, Jean-Christophe Janodet, and Frédéric Tantini. 2008. Learning Balls of Strings from Edit Corrections. Journal of Machine Learning Research 9, 8 (2008).

[8] Richard Beigel and William Gasarch. [n.d.]. A Proof that if $L = L_1 \cap L_2$ where L_1 is CFL and L_2 is Regular then L is Context Free Which Does Not use PDA's. http://www.cs.umd.edu/~gasarch/BLOGPAPERS/cfg.pdf

[9] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. 2014. Automata theory in nominal sets. Logical Methods in Computer Science 10 (2014).

[10] Janusz A Brzozowski. 1962. Canonical regular expressions and minimal state graphs for definite events. In Proc. Symposium of Mathematical Theory of Automata. 529–561.

[11] Janusz A Brzozowski. 1964. Derivatives of regular expressions. Journal of the ACM (JACM) 11, 4 (1964), 481–494.

[12] Janusz A. Brzozowski and Ernst Leiss. 1980. On equations for regular languages, finite automata, and sequential networks. Theoretical Computer Science 10, 1 (1980), 19–35.

[13] David Chiang. 2007. Hierarchical phrase-based translation. computational linguistics 33, 2 (2007), 201–228.

[14] David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter bounds on the expressivity of transformer encoders. In International Conference on Machine Learning. PMLR, 5544–5562. https://proceedings.mlr.press/v202/chiang23a/chiang23a.pdf

[15] Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 703–715.

[16] Noam Chomsky. 1959. On certain formal properties of grammars. Information and control 2, 2 (1959), 137–167.

[17] Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 541–553.

[18] Dingding Dong, Nitya Mani, and Yufei Zhao. 2023. On the number of error correcting codes. Combinatorics, Probability and Computing (2023), 1–14. https://doi.org/10.1017/S0963548323000111

[19] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming. 1–8.

[20] Philippe Duchon, Philippe Flajolet, et al. 2004. Boltzmann samplers for the random generation of combinatorial structures. Combinatorics, Probability and Computing 13, 4-5 (2004), 577–625.

[21] Jay Earley. 1970. An efficient context-free parsing algorithm. Commun. ACM 13, 2 (1970), 94–102.

[22] David Eppstein. 2014. k-best enumeration. arXiv preprint arXiv:1412.5075 (2014).

[23] Seymour Ginsburg and H Gordon Rice. 1962. Two families of languages related to ALGOL. Journal of the ACM (JACM) 9, 3 (1962), 350–371.

[24] Joshua Goodman. 1999. Semiring parsing. Computational Linguistics 25, 4 (1999), 573–606. https://aclanthology.org/J99-4004.pdf

[25] Vivek Gore, Mark Jerrum, Sampath Kannan, Z Sweedyk, and Steve Mahaney. 1997. A quasi-polynomial-time algorithm for sampling words from a context-free language. Information and Computation 134, 1 (1997), 59–74.

[26] Dick Grune and Ceriel J. H. Jacobs. 2008. Parsing as Intersection. Springer New York, New York, NY, 425–442. https://doi.org/10.1007/978-0-387-68954-8_13

[27] Matthew Hague, Artur Jeż, and Anthony W Lin. 2024. Parikh's Theorem Made Symbolic. Proceedings of the ACM on Programming Languages 8, POPL (2024), 1945–1977.

[28] Timothy Hickey and Jacques Cohen. 1983. Uniform random generation of strings in a context-free language. SIAM J. Comput. 12, 4 (1983), 645–655.

[29] Liang Huang and David Chiang. 2005. Better k-best parsing. In Proceedings of the Ninth International Workshop on Parsing Technology. 53–64.

[30] Liang Huang and David Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In Proceedings of the 45th annual meeting of the association of computational linguistics. 144–151.

[31] E. T. Irons. 1963. An Error-Correcting Parse Algorithm. Commun. ACM 6, 11 (nov 1963), 669–673. https://doi.org/10.1145/368310.368385

[32] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In Proceedings of the eighteenth international symposium on Software testing and analysis. 105–116.

[33] Lillian Lee. 2002. Fast context-free grammar parsing requires fast boolean matrix multiplication. Journal of the ACM (JACM) 49, 1 (2002), 1–15. https://arxiv.org/pdf/cs/0112018.pdf

[34] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, Vol. 10. 707–710. https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf

[35] Alexander K Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K Mansinghka. 2023. Sequential monte carlo steering of large language models using probabilistic programs. arXiv preprint arXiv:2306.03081 (2023).

[36] William Merrill, Ashish Sabharwal, and Noah A Smith. 2022. Saturated transformers are constant-depth threshold circuits. Transactions of the Association for Computational Linguistics 10 (2022), 843–856.

[37] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. ACM sigplan notices 46, 9 (2011), 189–195.

[38] Mark-Jan Nederhof and Giorgio Satta. 2004. The language intersection problem for non-recursive context-free grammars. Information and Computation 192, 2 (2004), 172–184.

[39] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. Software: Practice and Experience 25, 7 (1995), 789–810.

[40] Clemente Pasti, Andreas Opedal, Tiago Pimentel, Tim Vieira, Jason Eisner, and Ryan Cotterell. 2023. On the Intersection of Context-Free and Regular Languages. In Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, Andreas Vlachos and Isabelle Augenstein (Eds.). Association for Computational Linguistics, Dubrovnik, Croatia, 737–749. https://doi.org/10.18653/v1/2023.eacl-main.52

[41] Itiroo Sakai. 1961. Syntax in universal translation. In Proceedings of the International Conference on Machine Translation and Applied Language Analysis.

[42] Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, and Ranjit Jhala. 2022. Seq2Parse: neurosymbolic parse error repair. Proceedings of the ACM on Programming Languages 6, OOPSLA2 (2022), 1180–1206.

[43] Arto Salomaa. 1973. Formal languages. Academic Press, New York. 59–61 pages.

[44] Klaus U Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition 5 (2002), 67–85.

[45] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. Electronic Notes in Theoretical Computer Science 253, 7 (2010), 177–189.

[46] Kensen Shi, David Bieber, and Charles Sutton. 2020. Incremental sampling without replacement for sequence models. In International Conference on Machine Learning. PMLR, 8785–8795.

[47] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 620–635.

[48] Michalis K Titsias and Christopher Yau. 2017. The Hamming ball sampler. J. Amer. Statist. Assoc. 112, 520 (2017), 1598–1611.

[49] Leslie G Valiant. 1975. General context-free recognition in less than cubic time. Journal of computer and system sciences 10, 2 (1975), 308–315. http://people.csail.mit.edu/virgi/6.s078/papers/valiant.pdf

[50] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. Proceedings of the ACM on Programming Languages 9, POPL (2025), 1–32.

[51] Alexander William Wong, Amir Salimi, Shaiful Chowdhury, and Abram Hindle. 2019. Syntax and Stack Overflow: A methodology for extracting a corpus of syntax errors and fixes. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 318–322.

[52] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In International Conference on Machine Learning. PMLR, 11941–11952.

[53] Hao Zhang and Ryan McDonald. 2012. Generalized higher-order dependency parsing with cube pruning. In Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning. 320–331.

A LEVENSHTEIN AUTOMATA MATRICES

These are useful for visually checking different implementations.

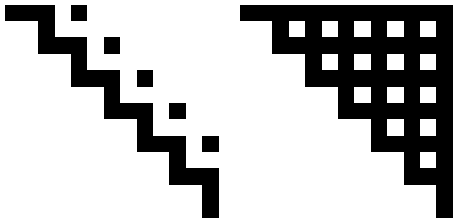


Fig. 19. Lev($|\sigma|$=6, Δ=1) adjacency and reachability matrices.

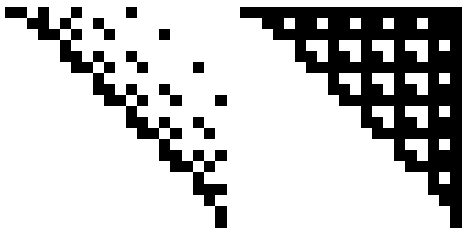


Fig. 20. Lev($|\sigma|$=6, Δ=2) adjacency and reachability matrices.

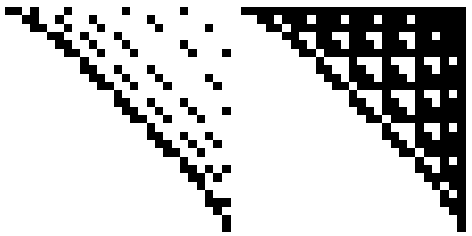


Fig. 21. Lev($|\sigma|$=6, Δ=3) adjacency and reachability matrices.

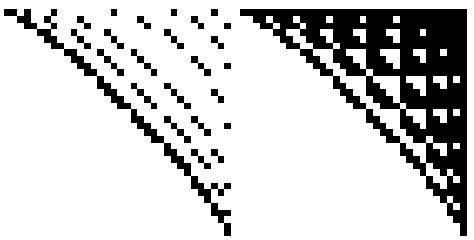


Fig. 22. Lev($|\sigma|$=6, Δ=4) adjacency and reachability matrices.

B LEVENSHTEIN AUTOMATA MINIMALITY

It is reasonable to ask whether the Levenshtein automaton defined is minimal, in the sense of whether there exists an automaton with fewer states than A yet still generates $\mathcal{L}(G_\cap)$ when intersected with $\mathcal{L}(G)$. In other words, given G and $\underset{\sim}{\sigma}$, is there an A' such that $|Q_{A'}| < |Q_A|$ yet $\mathcal{L}(G) \cap \mathcal{L}(A') = \mathcal{L}(G) \cap \mathcal{L}(A)$ still holds? In fact, there is a trivial example:

Theorem B.1. *Let $Q_{A'}$ be defined as $Q_A \setminus \{q_{n,0}\}$.*

Since $q_{n,0}$ accepts the original string $\underset{\sim}{\sigma}$ which is by definition outside $\mathcal{L}(G)$, we can immediately rule out this state. Moreover, we can define a family of automata with strictly fewer states than the full LBH construction by making the following observation: if we can prove one edit must occur before the last s tokens, we can rule out the last s states absorbing editless trajectories.

Theorem B.2. *$\varnothing = \mathcal{L}(\underset{\sim}{\sigma}_{1\ldots(n-s)} \cdot \Sigma^s) \cap \mathcal{L}(G)$ implies the states $[q_{n-i,0}]_{i \in 1\ldots s}$ are unnecessary.*

Likewise, if we expend our entire edit budget in the first p tokens, we will be unable to recover in a string where at least one repair must occur after the first p tokens.

Theorem B.3. *$\varnothing = \mathcal{L}(\Sigma^p \cdot \underset{\sim}{\sigma}_p) \cap \mathcal{L}(G)$ implies the states $[q_{i,d_{\max}}]_{i \in 0\ldots p}$ are unnecessary.*

Therefor, we can eliminate $p+s$ states from A by proving emptiness of $\mathcal{L}(\Sigma^p \cdot \underset{\sim}{\sigma}_{p\ldots(n-s)} \cdot \Sigma^s) \cap \mathcal{L}(G)$, without affecting $\mathcal{L}(G_\cap)$. Pictorially,

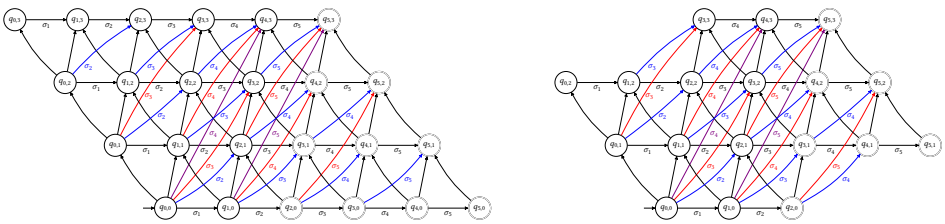


Fig. 23. Levenshtein NFA before and after pruning.

Pruned L-NFA for the broken string $\underset{\sim}{\sigma}$ = `[ ( + ) ]` with $G = \{S \to (S) \mid [S] \mid S + S \mid 1\}$.

`_ _ + ) ]` ❌ ∧ `_ _ _ ) ]` ✅

`[ ( + _ _` ❌ ∧ `[ ( _ _ _` ✅

C EXAMPLE REPAIRS

Below, we provide a few representative examples of broken code snippets and the corresponding human repairs that were successfully ranked first by our method. On the left is a complete snippet fed to the model and on the right, the corresponding human repair that was correctly predicted.

Original broken code	First predicted repair

`form sympy import *`
`x = Symbol('x', real=True)`
`x, re(x), im(x)`
	`from sympy import *`
	`x = Symbol('x', real=True)`
	`x, re(x), im(x)`

`result = yeald From(item.create())` `raise Return(result)`	`result = yield From(item.create())` `raise Return(result)`
`df.apply(lambda row: list(set(row['ids'))))`	`df.apply(lambda row: list(set(row['ids'])))`
`sum(len(v) for v items.values()))`	`sum(len(v) for v in items.values())`

`def average(values):`
`if values == (1,2,3):`
`return (1+2+3)/3`
`else if values == (-3,2,8,-1):`
`return (-3+2+8-1)/4`
	`def average(values):`
	`if values == (1,2,3):`
	`return (1+2+3)/3`
	`elif values == (-3,2,8,-1):`
	`return (-3+2+8-1)/4`

`dict = {`
`"Jan": 1`
`"January": 1`
`"Feb": 2 # and so on`
`}`
	`dict = {`
	`"Jan": 1,`
	`"January": 1,`
	`"Feb": 2 # and so on`
	`}`

`class MixIn(object)`
`def m():`
`pass`
 	
`class classA(MixIn):`
 	
`class classB(MixIn):`
	`class MixIn(object):`
	`def m():`
	`pass`
	
	`class classA(MixIn): pass`
	
	`class classB(MixIn): pass`

D RAW DATA

Raw data from Precision@k experiments across snippet length and Levenshtein distance from § 7.3. $|\sigma|$ indicates the snippet length and Δ indicates the Levenshtein distance between the broken and code and human fix computed over lexical tokens. For Tidyparse, we sample until exhausting the admissible set or a timeout of 30s is reached, whichever happens first, then rank the results. For the other models Precision@1, we sample one repair and report the percentage of repairs matching the human repair. For Precision@All, we report the percentage of repairs matching the human repair within the top 20000 samples.

	Δ	Precision@1							
$\lvert\sigma\rvert$		$(0, 10)$	$[10, 20)$	$[20, 30)$	$[30, 40)$	$[40, 50)$	$[50, 60)$	$[60, 70)$	$[70, 80)$
Tidyparse	1	0.56	0.44	0.43	0.49	0.55	0.55	0.53	0.57
	2	0.37	0.28	0.26	0.24	0.19	0.25	0.23	0.18
	3	0.18	0.20	0.19	0.15	0.10	0.09	0.11	0.11
Seq2Parse	1	0.35	0.41	0.40	0.37	0.31	0.29	0.27	0.21
	2	0.12	0.13	0.14	0.12	0.11	0.11	0.10	0.12
	3	0.03	0.07	0.08	0.09	0.09	0.02	0.07	0.06
BIFI	1	0.20	0.33	0.32	0.27	0.21	0.21	0.25	0.18
	2	0.18	0.18	0.21	0.19	0.19	0.18	0.11	0.11
	3	0.02	0.02	0.03	0.02	0.03	0.05	0.03	0.02
		Precision@All							
Tidyparse	1	1.00	1.00	1.00	0.99	0.99	1.00	0.97	0.97
	2	1.00	0.99	0.98	1.00	1.00	1.00	0.94	0.90
	3	1.00	0.98	0.80	0.70	0.55	0.42	0.42	0.31
BIFI	1	0.65	0.67	0.70	0.65	0.60	0.62	0.60	0.64
	2	0.52	0.41	0.37	0.32	0.27	0.27	0.21	0.24
	3	0.20	0.13	0.08	0.17	0.15	0.18	0.17	0.07

E SUPPLEMENTAL PROOFS

The problem of syntax error correction under a finite number of typographic errors is reducible to the bounded Levenshtein-CFL reachability problem, which can be formally stated as follows:

Definition E.1. The language edit distance (LED) is the minimum number of edits required to transform an invalid string into a valid one, where validity is defined as containment in a context-free language, ℓ, i.e., $\Delta^*(\underset{\sim}{\sigma}, \ell) \coloneqq \min_{\sigma \in \ell} \Delta(\underset{\sim}{\sigma}, \sigma)$, and Δ is the Levenshtein distance.

We seek to find the set of strings S such that $\forall \sigma' \in S, \Delta(\underset{\sim}{\sigma}, \sigma') \leq q$, where q is greater than or equal to the language edit distance. We call this set the *Levenshtein ball* of $\underset{\sim}{\sigma}$ and denote it $\Delta_q(\underset{\sim}{\sigma})$. Since $1 \leq \Delta^*(\underset{\sim}{\sigma}, \ell) \leq q$, we have $1 \leq q$. We now consider an upper bound on $\Delta^*(\underset{\sim}{\sigma}, \ell)$, i.e., the greatest lower bound on q such that $\Delta_q(\underset{\sim}{\sigma}) \cap \ell \neq \varnothing$.

Lemma E.2. *For any nonempty language ℓ and invalid string $\underset{\sim}{\sigma} : \Sigma^n \cap \bar{\ell}$, there exists an (σ', m) such that $\sigma' \in \ell \cap \Sigma^m$ and $0 < \Delta(\underset{\sim}{\sigma}, \ell) \leq \max(m, n) < \infty$.*

Proof. Since ℓ is nonempty, it must have at least one inhabitant $\sigma \in \ell$. Let σ' be the smallest such member. Since σ' is a valid sentence in ℓ, by definition it must be that $|\sigma'| < \infty$. Let $m \coloneqq |\sigma'|$. Since we know $\underset{\sim}{\sigma} \notin \ell$, it follows that $0 < \Delta(\underset{\sim}{\sigma}, \ell)$. Let us consider two cases, either $\sigma' = \varepsilon$, or $0 < |\sigma'|$:

- If $\sigma' = \varepsilon$, then $\Delta(\underset{\sim}{\sigma}, \sigma') = n$ by full erasure of $\underset{\sim}{\sigma}$, or
- If $0 < m$, then $\Delta(\underset{\sim}{\sigma}, \sigma') \leq \max(m, n)$ by overwriting.

In either case, it follows $\Delta(\underset{\sim}{\sigma}, \ell) \leq \max(m, n)$ and ℓ is always reachable via a finite nonempty set of Levenshtein edits, i.e., $0 < \Delta(\underset{\sim}{\sigma}, \ell) < \infty$. □

Let us now consider the maximum growth rate of the *admissible set*, $\ell_\cap \coloneqq \Delta_q(\underset{\sim}{\sigma}) \cap \ell$, as a function of q and n. Let $\bar{\ell} \coloneqq \{\underset{\sim}{\sigma}\}$. Since $\bar{\ell}$ is finite and thus regular, $\ell = \Sigma^* \setminus \{\underset{\sim}{\sigma}\}$ is regular by the closure of regular languages under complementation, and thus context-free a fortiori. Since ℓ accepts every string except $\underset{\sim}{\sigma}$, it represents the worst CFL in terms of asymptotic growth of $\ell_\cap$.

Lemma E.3. *The complexity of enumerating $\ell_\cap$ is upper bounded by $\mathcal{O}\left(\sum_{c=1}^{q} \binom{cn+n+c}{c} (|\Sigma| + 1)^c\right)$.*

Proof. We can overestimate the size of $\ell_\cap$ by considering the number of unique ways to insert, delete, or substitute c terminals into a string $\underset{\sim}{\sigma}$ of length n. This can be overaproximated by interleaving ε^c around every token, i.e., $\underset{\sim}{\sigma}_\varepsilon \coloneqq (\varepsilon^c \underset{\sim}{\sigma}_i)_{i=1}^n \varepsilon^c$, where $|\underset{\sim}{\sigma}_\varepsilon| = cn + n + c$, and only considering substitution. We augment $\Sigma_\varepsilon \coloneqq \Sigma \cup \{\varepsilon\}$ so that deletions and insertions may be treated as special cases of substitution. Thus, we have $cn + n + c$ positions to substitute ($|\Sigma_\varepsilon|$) tokens, i.e., $\binom{cn+n+c}{c} |\Sigma_\varepsilon|^c$ ways to edit $\underset{\sim}{\sigma}_\varepsilon$ for each $c \in [1, q]$. This upper bound is not tight, as overcounts many identical edits w.r.t. $\underset{\sim}{\sigma}$. Nonetheless, it is sufficient to show $|A| < \sum_{c=1}^{q} \binom{cn+n+c}{c} |\Sigma_\varepsilon|^c$. □

We note that the above bound applies to all strings and languages, and relates to the Hamming bound on $H_q(\underset{\sim}{\sigma}_\varepsilon)$, which only considers substitutions. In practice, much tighter bounds may be obtained by considering the structure of ℓ and $\underset{\sim}{\sigma}$. For example, based on an empirical evaluation from a dataset of human errors and repairs in Python code snippets ($|\Sigma| = 50, |\underset{\sim}{\sigma}| < 40, \Delta(\underset{\sim}{\sigma}, \ell) \in [1, 3]$), we estimate the *filtration rate*, i.e., the density of the admissible set relative to the Levenshtein ball, $D = |A|/|\Delta_q(\underset{\sim}{\sigma})|$ to have empirical mean $E_\sigma[D] \approx 2.6 \times 10^{-4}$, and variance $\mathrm{Var}_\sigma[D] \approx 3.8 \times 10^{-7}$.