Syntax Repair as Language Intersection

BREANDAN CONSIDINE

We introduce a new technique for repairing syntax errors in arbitrary context-free languages. This technique models syntax repair as a language intersection problem by defining a finite language that provably generates every syntactically valid repair within a given edit distance. Leveraging a theoretical connection between the Bar-Hillel construction from formal language theory and CFL reachability from program analysis, we show that repairability in a finite number of typographic edits is polylogarithmic parallel time decidable and provide an enumeration algorithm based on the Brzozowski derivative. Finally, we evaluate this algorithm and its implementation, demonstrating state-of-the-art results on a Python syntax repair benchmark.

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 short list of suggestions that anticipate with high probability the author's intended 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, specialized parsers with error recovery rules are used to propose a single, least-cost repair. While appealing for their interpretability and well-understood algorithmic properties, these methods are unable to handle ambiguity and must rely on relatively brittle heuristics to select the repair.

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, these methods typically use some form of approximate inference and thus require expensive postprocessing or rejection sampling to ensure the generated results conform to the grammar.

The primary shortcoming with both approaches is *they generate far too few repairs.* As we will show, even if the repair model guarantees correctness or has good statistical generalization properties, it is likely to miss the intended repair in ambiguous scenarios or when there are many candidates from which to choose. Most syntax errors, however, require only a few typographic modifications 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 can we quickly recover the most probable repairs in proximity to a syntactically broken code snippet? To address this problem, we propose to extensively evaluate the probability of every repair within a fixed edit distance. At first, this might seem to take much longer than generating a single repair, but if we intend to quickly generate probable repairs and not just valid ones, extensive search becomes highly advantageous. To ensure the search space is well-defined, we will construct and decode a regular expression that generates all and only valid repairs within a fixed edit distance, thereby avoiding rejection sampling entirely without skipping any nearby valid repairs. This construction is shown in Fig. 1.
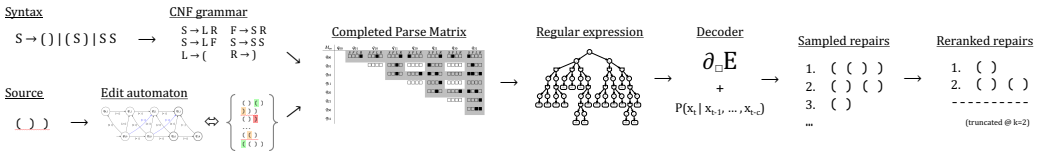

Fig. 1. Our algorithm first constructs an automaton representing all strings within a certain edit distance. This automaton is parsed into a matrix denoting all valid repairs in the programming language and edit distance. We construct a regular expression (RE) from the matrix, and finally decode the RE using an n-gram model to produce a finite list of samples, then rerank and truncate this list to obtain our final repairs.

To operationalize this technique, we design, develop and benchmark a new developer tool for syntax repair which is readily executable on multicore CPUs and GPUs. We provide a reference implementation of our tool and show these parallel computing resources, which typically sit idle during text editing, can be profitably used to accelerate real-time program repair.

Finally, we show the efficacy of this technique for locating and repairing syntax errors of up to three edits and eighty lexical tokens in under ten seconds, practical for a few lines of source code in realistic programming languages. Our work shows this technique is highly effective at predicting the human repair across a dataset of Python source code, up to 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 ($P \colon V \rightarrow (V \mid \Sigma)^{+}$), and a start symbol, ($S$). Every CFG is reducible to so-called *Chomsky Normal Form* [15], $P' \colon V \rightarrow (V^2 \mid \Sigma)$, where every production is either (1) a binary production $w \rightarrow xz$, or (2) a unit production $w \rightarrow t$, where $w, x, z : V$ and $t : \Sigma$. For example:

$$G = \big\{\, S \rightarrow S\,S \mid (\,S\,) \mid (\,) \,\big\} \Longrightarrow G' = \big\{\, S \rightarrow Q\,R \mid S\,S \mid L\,R, \quad R \rightarrow), \quad L \rightarrow (, \quad Q \rightarrow L\,S \,\big\}$$

Likewise, a finite state automaton (FSA) is a quintuple $\mathcal{A} = \langle Q, \Sigma, \delta, q_\alpha, F \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, q_α is the initial state, and $F \subseteq Q$ are the accepting states. 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) fragment containing concatenation, conjunction and disjunction:

Definition 2.1 (Star-free GRE fragment). Let $e : E$ be an expression defined by the grammar:

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

where ε is the empty symbol. Semantically, we interpret these expressions as denoting languages:

$$\mathcal{L}(\varnothing) = \varnothing \qquad\qquad \mathcal{L}(\,x \cdot z\,) = \mathcal{L}(x) \circ \mathcal{L}(z)^{[1]}$$
$$\mathcal{L}(\,\varepsilon\,) = \{\varepsilon\} \qquad\qquad \mathcal{L}(x \vee z) = \mathcal{L}(x) \cup \mathcal{L}(z)$$
$$\mathcal{L}(\,a\,) = \{a\} \qquad\qquad \mathcal{L}(x \wedge z) = \mathcal{L}(x) \cap \mathcal{L}(z)$$

Brzozowski [9] introduces an operator, $\partial : E \times \Sigma \rightarrow E$, which quotients a language by some prefix,

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

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

---

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

Primarily, this gadget was designed to handle membership queries, for which purpose it has found a number of applications [38, 49, 53] 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:*

$$\partial_\sigma(e) : E \to E = \begin{cases} e & \textit{if } \sigma = \varepsilon \\ \partial_b(\partial_a e) & \textit{if } \sigma = a \cdot b, a \in \Sigma, b \in \Sigma^* \end{cases} \tag{1}$$

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 $\sigma \in \mathcal{L}(e)$:*

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

$$\texttt{choose}(e) : E \to \Sigma^+ = \begin{cases} e & \textit{if } e \in \Sigma \\ \big(s \leftsquigarrow \texttt{follow}(e)\big) \cdot \texttt{choose}(\partial_s e) & \textit{if } e = x \cdot z \\ \texttt{choose}\big(e' \leftsquigarrow \{x, z\}\big) & \textit{if } e = x \vee z \end{cases} \tag{3}$$

Here, we use the $\leftsquigarrow$ 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:

$$P(e) : E \to \big(V \to (\Sigma \mid V \mid V^2)\big) = \begin{cases} \{S_e \to e\} & \text{if } e \in \Sigma \\ P(x) \cup P(z) \cup \{S_e \to S_x S_z\} & \text{if } e = x \cdot z \\ P(x) \cup P(z) \cup \{S_e \to S_x, S_e \to S_z\} & \text{if } e = x \vee z \end{cases} \tag{4}$$

where the CFG is $G(e) = \langle V, \Sigma, P(e), S_e \rangle$ with V being nonterminals in $P(e)$. Therefore, 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 CFG, $G = \langle V, \Sigma, P, S \rangle$, and nondeterministic finite automata (NFA), $A = \langle Q, \Sigma, \delta, q_\alpha, F \rangle$, there is a CFG, $G_\cap = \langle V_\cap, \Sigma_\cap, P_\cap, S_\cap \rangle$ s.t. $\mathcal{L}(G_\cap) = \mathcal{L}(G) \cap \mathcal{L}(A)$.*

Salomaa [47] introduces a direct, but inefficient construction for the intersection grammar:

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

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

however, most synthetic productions in $P_\cap$ will be non-generating or unreachable. This method will construct a synthetic production for state pairs that are not even connected by any path, which is clearly excessive. In § 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, $t : \Sigma_U^* \to \Sigma_G^*$. In the compiler nomenclature, t is called a *lexer* and would typically be regular under mild conditions. In this paper, we do not consider t 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 intended 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.
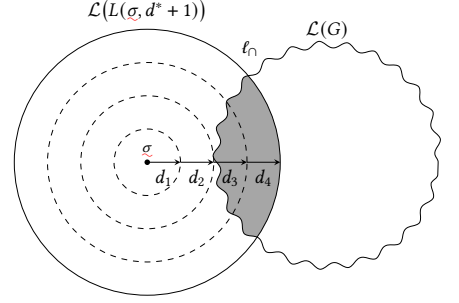

Fig. 2. CFL intersection with the local edit region around a broken code snippet, where $d^* = 3$ is the language edit distance (LED).

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 intersection language, $\ell_\cap$, typically contains a large number of possible repairs, we want a procedure that surfaces both 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 $P_\theta : \Sigma^* \to [0, 1] \subset \mathbb{R}$, find the top-$k$ maximum probability repairs. That is,

$$R(\ell_\cap, P_\theta) : 2^{\Sigma^*} \times (\Sigma^* \to \mathbb{R}) \to (\Sigma^*)^{\leq k} = \underset{\boldsymbol{\sigma} \subseteq \ell_\cap, |\boldsymbol{\sigma}| \leq k}{\mathrm{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} P_\theta(\sigma) \tag{5}$$

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 $\sigma \leftsquigarrow \ell_\cap$, 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 $\sigma \leftsquigarrow \Sigma^*$, as do most neural language models.

As we will demonstrate, the ranked repair problem can be factorized into three subproblems: (1) exact representation, (2) decoding and (3) reranking. Instead of working with strings, we will explicitly construct a grammar which soundly and completely generates $\ell \cap \mathcal{L}\big(L(\underset{\sim}{\sigma}, d)\big)$, then decode repairs from its language. By ensuring decoding is sufficiently precise and extensive, ensuring the retrieved set contains the true repair can be achieved with a much simpler, syntax-oblivious model. Finally, we will train a language model to rerank the repair candidates and take the top-k results.

3 METHOD

The key to solving this problem is to treat finite language intersections as matrix exponentiation, exploiting a correspondence between the Bar-Hillel construction and CFL reachability. We show that if one of the participants in the language intersection is presented as an acyclic FSA, the finite intersection nonemptiness problem is polylogarithmic parallel time decidable. Formally,

THEOREM 3.1. *For any CFG, $G = \langle V, \Sigma, P, S \rangle$, and acyclic NFA (ANFA), $A = \langle Q, \Sigma, \delta, q_\alpha : Q, F \subseteq Q \rangle$, there exists a decision procedure $\Psi : CFG \to ANFA \to \mathbb{B}$ such that $\Psi(G, A) \models [\mathcal{L}(G) \cap \mathcal{L}(A) \neq \varnothing]$ requiring $\mathcal{O}(\log^2 |Q| + \log |Q||V|)$ time using $\mathcal{O}(|Q|^2|V|)$ parallel random access (PRAM) processors.*

PROOF. To prove nonemptiness, we must show there exists a path $q_\alpha \rightsquigarrow q_\omega$ in A such that $q_\omega : F$ where $q_\alpha \rightsquigarrow q_\omega \vdash S$. At least one of 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 $\exists a.\left((p \xrightarrow{a} r) \in \delta \wedge (w \to a) \in P\right)$, 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). Note Q can be ordered topologically in $\mathcal{O}(\log^2 |Q|)$ time [16] using matrix multiplication. We initialize M thusly:

$$M_0[r, c, w] = \bigvee_{a \in \Sigma} \left\{ a \mid (w \to a) \in P \wedge (q_r \xrightarrow{a} q_c) \in \delta \right\} \tag{6}$$

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

$$[\ell \oplus r]_w = [\ell_w \vee r_w] \quad \text{and} \quad [\ell \otimes r]_w = \bigvee_{x, z \in V} \left\{ \ell_x \cdot r_z \mid (w \to xz) \in P \right\}. \tag{7}$$

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

$$\exp(M) = \sum_{i=0}^{\infty} M_0^i = \sum_{i=0}^{|Q||V|} M_0^i \text{ (since } M_0 \text{ is SUT and thus nilpotent)}. \tag{8}$$

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

$$\sum_{i=0}^{2n} M_0^i = T(2n) = \begin{cases} M_0, & \text{if } n = 1, \\ T(n) + T(n)^2 & \text{otherwise.} \end{cases} \tag{9}$$

Therefore, 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, $S_\cap$.

$$S_\cap = \bigvee_{q_\omega \in F} \exp(M)[q_\alpha, q_\omega, S], \text{ and } \Psi = [S_\cap \neq \varnothing]. \tag{10}$$

Note that it is possible to check Ψ before each recurrence of T and escape immediately thereafter in the positive case. Optimistically, this can occur in $\Omega(\log_2 p^*)$ time, where p^* is the length of the shortest path through δ, $p^* = \min_{q_\omega \in F} |q_\alpha \rightsquigarrow q_\omega|$. In case of nonemptiness, one may simply choose($S_\cap$) (see Eq. 3) to decode a witness $\sigma \in \mathcal{L}(G) \cap \mathcal{L}(A)$. In either case, the decision procedure terminates in $\mathcal{O}(\log^2 |Q| + \log |Q||V|)$ parallel time with $\mathcal{O}(|Q|^2|V|)$ processors. □

---

[2]Customarily, there is a $\frac{1}{k!}$ factor to modulate exploding values, 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 intersection with a singleton language. Second, we will introduce *completion* as an intersection that admits terminal wildcards in fixed locations. Thirdly, we consider syntax repair, where we will intersect a language representing all possible edit paths within a certain distance to determine the location(s) and fill them with the appropriate terminal(s).

4.1 Recognition as intersection

In the case of ordinary CFL recognition, the automaton forms a single row and accepts one word:
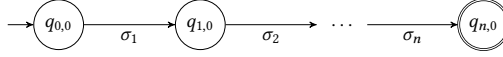
Since the word is predetermined, we just need to keep track of nonterminal subsets for each substring. So, given a CFG, $G' : \mathcal{G}$ in Chomsky Normal Form (CNF), we can construct a recognizer for strings $\sigma : \Sigma^n$ as follows. Let 2^V be our domain, 0 be $\varnothing$, $\oplus$ be $\cup$, and $\otimes$ be defined as:

$$X \otimes Z = \big\{ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \big\} \tag{11}$$

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:

$$M_0 = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \varnothing & \cdots & \varnothing \\ & & \ddots & & \varnothing \\ & & & \ddots & \hat{\sigma}_n \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix}, \; M_1 = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \Lambda & \cdots & \varnothing \\ & & \ddots & & \Lambda \\ & & & \ddots & \hat{\sigma}_n \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix}, \ldots, M_\infty = \begin{pmatrix} \varnothing & \hat{\sigma}_1 & \Lambda & \cdots & \Lambda_\sigma^* \\ & & \ddots & & \Lambda \\ & & & \ddots & \hat{\sigma}_n \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix} \tag{12}$$

Once the fixpoint M_∞ is attained, the proposition $[S \in \Lambda_\sigma^*]$ [3] decides language membership, i.e., $[\sigma \in \mathcal{L}(G)]$. So far, this procedure is essentially the textbook CYK algorithm in a linear algebraic notation [25] and a well-established technique in the parsing literature [26].

4.2 Completion as intersection

Let us now consider a problem of intermediate difficulty, wherein we are given a string template admitting edits at fixed locations, which can be filled by any terminal. When intersected with a CFL, this specifies a finite language whose contents are the set of all words consistent with the template. This problem we call *completion*. Formally,

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 members $\{\sigma' : \Sigma^+ \mid \sigma' \sqsubseteq \sigma\}$ as $\mathrm{H}(\sigma)$. Given a *porous string*, $\sigma : \underline{\Sigma}^+$ we seek all syntactically valid members, 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.:

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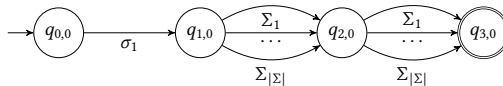
[3]Hereinafter, we use Iverson brackets to denote the indicator function of a predicate with free variables, i.e., $[P] \Leftrightarrow \mathbb{1}(P)$.

This corresponds to a template with two holes, $\sigma = 1 \_ \_$. Suppose the context-free grammar is $G = \{S \rightarrow NON, O \rightarrow + \mid \times, N \rightarrow 0 \mid 1\}$. This grammar will first be rewritten into CNF as $G' = \{S \rightarrow NL, N \rightarrow 0 \mid 1, O \rightarrow \times \mid +, L \rightarrow ON\}$. 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		$\mathbb{Z}_2^{|V|}$		GRE$^{|V|}$	
M_0	$\{N\}$ $\quad$ $\{N,O\}$ $\qquad$ $\{N,O\}$		$\substack{L\,N\,O\,S \\ \square\blacksquare\square\square}$ $\quad$ $\square\blacksquare\blacksquare\square$ $\qquad$ $\square\blacksquare\blacksquare\square$		$E_{0,1}$ $\quad$ $E_{1,2}$ $\qquad$ $E_{2,3}$	
M_1	$\{N\}$ $\quad \varnothing$ $\quad$ $\{N,O\}$ $\quad \{L\}$ $\qquad$ $\{N,O\}$		$\square\blacksquare\square\square$ $\quad \square\square\square\square$ $\quad \square\blacksquare\blacksquare\square$ $\quad \blacksquare\square\square\square$ $\qquad \square\blacksquare\blacksquare\square$		$E_{0,1}$ $\quad E_{0,2}$ $\quad E_{1,2}$ $\quad E_{1,3}$ $\qquad E_{2,3}$	
M_2 $=$ M_∞	$\{N\}$ $\quad \varnothing$ $\quad \{S\}$ $\quad$ $\{N,O\}$ $\quad \{L\}$ $\qquad$ $\{N,O\}$		$\square\blacksquare\square\square$ $\quad \square\square\square\square$ $\quad \square\square\square\blacksquare$ $\quad \square\blacksquare\blacksquare\square$ $\quad \blacksquare\square\square\square$ $\qquad \square\blacksquare\blacksquare\square$		$E_{0,1}$ $\quad E_{0,2}$ $\quad E_{0,3}$ $\quad E_{1,2}$ $\quad E_{1,3}$ $\qquad E_{2,3}$	

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., $\big[S \in M_\infty[0,3]\big] \Leftrightarrow \big[M_\infty[0,3,3] = \blacksquare\big] \Leftrightarrow \big[A(\sigma) \neq \varnothing\big]$. 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 $X \otimes Z = [X_2 \cdot Z_1, \varnothing, \varnothing, X_1 \cdot Z_0]$ and $X \oplus Z = [X_i \vee Z_i]_{i \in [0,|V|)}$, 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 $\otimes$. To solve for M_∞, we proceed by first computing $E_{0,2}, E_{1,3}$:

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

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

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

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

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 $\mathtt{choose}\big(1 \cdot \{+, \times\} \cdot \{0, 1\}\big)$ (see Eq. 3).

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 [48] contains cycles and ε-transitions, we propose a variant which is ε-free and acyclic. Furthermore, we adopt a symbolic form that 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

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

unlabeled lattice. The following construction is equivalent to Schultz and Mihov's original Levenshtein automaton, but is amenable to our purposes as it does not contain any ε-arcs, and instead uses skip connections to recognize consecutive deletions of varying lengths.
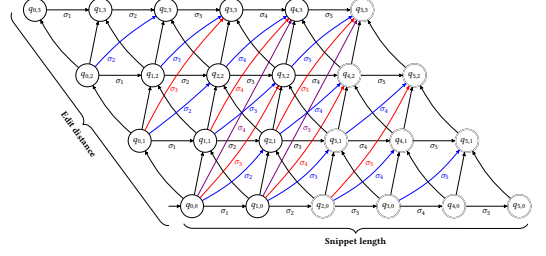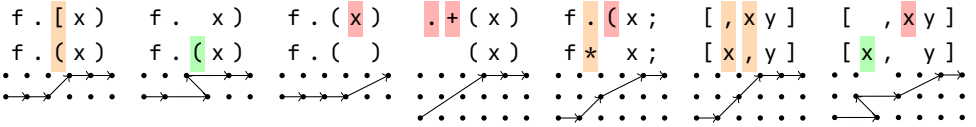
$$\frac{s \in \Sigma \quad i \in [0, n] \quad j \in [1, d_{\max}]}{(q_{i,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \searrow \qquad \frac{s \in \Sigma \quad i \in [1, n] \quad j \in [1, d_{\max}]}{(q_{i-1,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nearrow$$

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

$$\frac{}{q_{0,0} \in I} \text{ INIT} \qquad \frac{q_{i,j} \in Q \quad |n - i + j| \le d_{\max}}{q_{i,j} \in F} \text{ DONE}$$

Each type of arc plays a specific role. $\searrow$ handles insertions, $\nearrow$ handles substitutions and $\cdots$ handles deletions of one or more terminals. Let us consider some illustrative cases.

```
f . [ x )      f .   x )      f . ( x )      . + ( x )      f . ( x ;      [ , x y ]      [   , x y ]
f . ( x )      f . ( x )      f . ( )        ( x )          f * x ;        [ x , y ]      [ x ,   y ]
```

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 $\Delta(\sigma, \sigma') \le d_{\max}$.

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

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

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

Using a symbolic predicate in the construction of the NFA avoids the unnecessary creation of $2(|\Sigma| - 1) \cdot |\sigma| \cdot d_{\max}$ arcs and reduces the representation size of the resulting automaton, but does not affect the underlying semantics. Since negation is only permitted over terminals and never propagates further, complementation can be stricken from the usual GRE axioms (Theorem 2.1).

As a concrete example, suppose we have the string, $\tilde{\sigma} = \texttt{()}$ 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, $G' = \{S \rightarrow LR, S \rightarrow LF, S \rightarrow SS, F \rightarrow SR, L \rightarrow (, R \rightarrow)\}$ 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 constrains 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 $\text{LED}(\sigma, G) = 1$. 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 $\oplus, \otimes$ over GRE vectors, where $X \otimes Z = [X_x \cdot Z_z \mid (w \rightarrow xz) \in P]_{w \in V}$ and $X \oplus Z = [X_w \vee Z_w]_{w \in V}$. 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 $S_\cap = q_{00}Sq_{20} \vee q_{00}Sq_{31}$ shown in Fig. 9. Once this regex is constructed, decoding becomes simply a matter of invoking $\texttt{choose}(S_\cap)$. In this case, there are only a few choices, but in general, there can be a vast multitude.
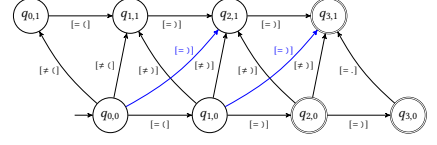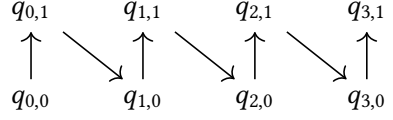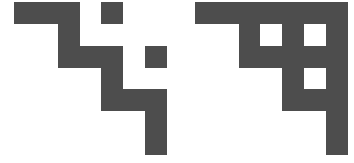

Fig. 4. Simple Levenshtein automaton.

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

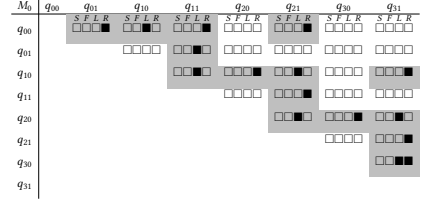Fig. 6. Adjacency and reachability matrix.
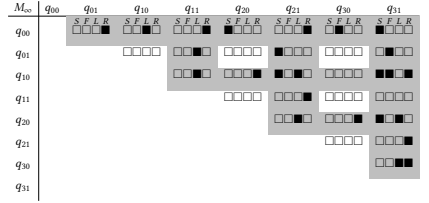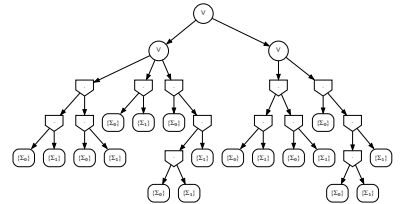

Fig. 7. Initial parse chart configuration.

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 shall start with a few cursory but illuminative approaches, then proceed towards a more refined solution.

5.1 Mode collapse

Ordinarily, one might think to train a top-down PCFG sampler using a treebank of well-formed code snippets, however this method is highly degenerate in the finite case, exhibiting poor sample diversity. Consider an illustrative pathological case for top-down ancestral (TDA) sampling:

$$G = \left\{ S \rightarrow A\,B \left(\frac{10^5 - 1}{10^5} \right), S \rightarrow C\,C \left(\frac{1}{10^5} \right), A \rightarrow a\,(1), B \rightarrow b\,(1), C \rightarrow a \left(\frac{1}{26} \right) \mid \ldots \mid z \left(\frac{1}{26} \right) \right\}$$

Such a sampler will almost always yield ab, but most of $\mathcal{L}(G)$ is concealed in the hidden branch, $S \rightarrow CC$. Though a contrived example, it illustrates why TDA sampling is unviable: our sampler should match 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 simply generate every tree. While the whole set can be materialized in some cases when the intersection language is small, this strategy is clearly suboptimal in the worst-case. Nevertheless, it is useful for checking completeness.

THEOREM 5.1 (ENUMERATION). *We will first denote the number of unique trees in a regex as $|e|$:*

$$|e| : E \rightarrow \mathbb{N} = \begin{cases} 1 & if\,e \in \Sigma \\ x \times z & if\,e = x \cdot z \\ x + z & if\,e = x \vee z \end{cases} \tag{13}$$

To enumerate, we construct a bijection between trees and integers, then invoke $\bigcup_{i=0}^{|R|} \{enum(R, i)\}$:

$$enum(e, n) : E \times \mathbb{N} \rightarrow \Sigma^* = \begin{cases} e & if\,e \in \Sigma \\ enum\left(x, \lfloor \frac{n}{|z|} \rfloor\right) \cdot enum\left(z, n \bmod |z|\right) & if\,e = x \cdot z \\ enum\left((x, z)_{\min(1, \lfloor \frac{n}{|x|} \rfloor)}, n - |x| \min(1, \lfloor \frac{n}{|x|} \rfloor)\right) & if\,e = x \vee z \end{cases} \tag{14}$$

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

5.3 The problem with ambiguity

The main problem with the previous approach is that it counts distinct trees, which overcounts the total number of words, $|\mathcal{L}(G_\cap)|$. Since the Levenshtein automaton can be ambiguous, this causes certain repairs to be overrepresented, resulting in a pernicious bias. Consider, for example,

LEMMA 5.2. *If the FSA, α, is ambiguous, then the intersection grammar, $G_\cap$, can be ambiguous.*

PROOF. Let ℓ be the language defined by $G = \{S \rightarrow LR, L \rightarrow (, R \rightarrow)\}$, where $\alpha = L(\sigma, 2)$, the broken string σ 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 $S \rightarrow q_{00}Lq_{21}\,q_{21}Rq_{22}$, and () with the parse $S \rightarrow q_{00}Lq_{11}\,q_{11}Rq_{22}$. □

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

5.4 Disambiguation

To count the number of distinct repairs, we will need to convert $G_\cap$ to an automaton. Since $\mathcal{L}(G_\cap)$ is finite, it must be regular a fortiori. Recalling the definition for an NFA, $\langle Q, \Sigma, \delta, q_\alpha : Q, F \subseteq Q \rangle$, and star-free regex, $e \rightarrow \Sigma \mid e \vee e \mid e \wedge e$, we will proceed by structural induction on the regex:

$$N(e) = \begin{cases} \langle \{q_\alpha, q_\omega\} & , \{q_\alpha \xrightarrow{e} q_\omega\} & , q_\alpha, \{q_\omega\} \quad \rangle \text{ if } e \in \Sigma \\[4pt] \langle Q_x \cup Q_z & , \{q \xrightarrow{s} q_{\alpha z} \mid (q \xrightarrow{s} q_\omega^{\in F_x}) \in \delta_x\} \cup \delta_x \cup \delta_z & , q_{\alpha x}, F_z \quad \rangle \text{ if } e = x \cdot z \\[4pt] \begin{aligned} Q_x \cup \{q_{\alpha e}\} \cup \\ Q_z \cup \{q_{\omega e}\} \end{aligned} & , \begin{aligned} \{q_{\alpha e} \xrightarrow{s} q \mid (q_{\alpha x, \alpha z} \xrightarrow{s} q) \in \delta_{x,z}\} \cup \delta_x \cup \\ \{q \xrightarrow{s} q_{\omega e} \mid (q \xrightarrow{s} q_\omega^{\in F_{x,z}}) \in \delta_{x,z}\} \cup \delta_z \end{aligned} & , q_{\alpha e}, \{q_{\omega e}\} \quad \rangle \text{ if } e = x \vee z \\ \multicolumn{3}{c}{\text{------------ or ------------}} \\ \langle Q_x \cup Q_z \cup \{q_{\alpha e}\}, \{q_{\alpha e} \xrightarrow{s} q \mid (q_{\alpha x, \alpha z} \xrightarrow{s} q) \in \delta_{x,z}\} \cup \delta_x \cup \delta_z, q_{\alpha e}, F_x \cup F_z \rangle \text{ if } e = x \vee z \end{cases}$$

Though less conventional than Thompson's construction, $N(e)$ avoids the creation of unnecessary ε arcs. And while slightly more verbose, we find the topology induced by the first version of the $\vee$ case to be more favorable for minimization. Continuing with our running example from § 4.3, we will use Brzozowski's algorithm [8] to construct the unique minimal DFA, $D_\cap^* \equiv_\mathcal{L} G_\cap$:
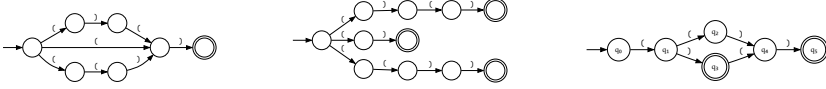

Fig. 10. FSA for $\mathcal{L}(L((\,)), 1) \cap \mathcal{L}(G')$ (a) with or (b) without $\vee$-merging, and then (c) post-minimization.

Since $\mathcal{L}(G_\cap)$ is necessarily finite, we can infer that the corresponding DFA is acyclic and thus representable as an upper triangular adjacency matrix under a topological ordering of δ. For any such DFA, we can ascertain the size of its language by counting walks from q_α to $q_\omega \in F$. Letting A be the adjacency matrix for $D_\cap^*$, i.e., $A[q, q'] = \big[1 \text{ if } \exists s : \Sigma \text{ s.t. } (q \xrightarrow{s} q') \in \delta \text{ else } 0 \big]$, the number of words it recognizes is given via the transfer matrix method [23], that is,

$$C(A, q_\alpha, F) : \mathbb{N}^{|Q| \times |Q|} \times Q \times 2^Q \rightarrow \mathbb{N} = \sum_{q_\omega \in F} (I - A)^{-1}[q_\alpha, q_\omega] = \sum_{q_\omega \in F} \sum_{i=0}^{|Q|-1} A^i[q_\alpha, q_\omega] \tag{15}$$

Plugging in powers of the adjacency matrix for the DFA shown in Fig. 10.(c), we arrive at the total:

$$(I - A)^{-1} = \qquad I + A \qquad + \qquad A^2 \qquad + \qquad A^3 \qquad + \qquad A^4 \tag{16}$$

$$= \begin{pmatrix} 1 & 1 & & & & \\ & 1 & 1 & 1 & & \\ & & 1 & & 1 & \\ & & & 1 & 1 & \\ & & & & 1 & 1 \\ & & & & & 1 \end{pmatrix} + \begin{pmatrix} & 1 & 1 & & & \\ & & & 2 & & \\ & & & & 1 & \\ & & & & & 1 \\ & & & & & \\ & & & & & \end{pmatrix} + \begin{pmatrix} & & & 2 & & \\ & & & & 2 & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix} + \begin{pmatrix} & & & & 2 & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix} \tag{17}$$

$$= \begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 \\ & 1 & 1 & 1 & 2 & 2 \\ & & 1 & & 1 & 1 \\ & & & 1 & 1 & 1 \\ & & & & 1 & 1 \\ & & & & & 1 \end{pmatrix} \text{ therefore, } \big| \mathcal{L}(D_\cap^*) \big| = C\big(A, q_0, \{q_3, q_5\}\big) = \underline{1} + \underline{2} = 3. \tag{18}$$

Note the model counting problem for arbitrary GREs is strictly harder than deciding intersection nonemptiness as it requires determinization, however, weak bounds may be obtained by applying C to the FSA generated by $N(e)$ or by direct analysis of e. While the inequality $C_{D_\cap^*} \leq C_{N(e)} \leq |e|$ will hold, the bounds provided by the latter approximations may be vacuous, whereas $C_{D_\cap^*}$ is exact.

6 IMPLEMENTATION

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

(1) lev_build : $\Sigma^n \times \mathbb{N} \to$ 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 $\to$ GRE – constructs the regular expression for $G_\cap$.
(4) reg_dcode : GRE $\times \mathbb{N}^{|\Sigma|^{c \approx 4}} \times \mathbb{N} \to (\Sigma^+)^{k \approx 10}$ – 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 § 4.3.

Algorithm 1 lev_build pseudocode

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

4:

$$\delta \leftarrow \delta \cup \begin{cases} q_{h,i} \overset{[\neq \sigma_{j+1}]}{\to} q_{j,k} & \text{if } h = j & \wedge\, i = k - 1 & \searrow \\ q_{h,i} \overset{[\neq \sigma_j]}{\to} q_{j,k} & \text{if } h = j - 1 & \wedge\, i = k - 1 & \nearrow \\ q_{h,i} \overset{[=\sigma_j]}{\to} q_{j,k} & \text{if } h = j - 1 & \wedge\, i = k & \rightarrow \\ q_{h,i} \overset{[=\sigma_j]}{\to} q_{j,k} & \text{if } 1 \le j - h - 1 \le d_{\max} \wedge 1 \le k - i \le d_{\max} & \cdots \end{cases}$$

5:         $Q \leftarrow Q \cup \{q_{h,i}, q_{j,k}\}$
6:     $I \leftarrow \{q_{0,0}\}, F \leftarrow \{q_{i,j} \mid n - i + j \le d_{\max}\}$
7:     **return** $\langle Q, \Sigma, \delta \subseteq Q \times (\Sigma \to \mathbb{B}) \times Q, q_\alpha, F \rangle$ ▷ Returns a [symbolic] 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 language).

1: **procedure** cfl_fixpt($\langle \Sigma, V, P, S \rangle$: CFG, $\langle Q, \Sigma, \delta, q_\alpha, F \rangle$: NFA)
2:     $R : \mathbb{B}^{|Q| \times |Q|} \leftarrow \big[\blacksquare \text{ if } \exists \sigma \in \Sigma^+ \mid q \overset{\sigma}{\leadsto} q' \text{ else } \square \big]_{q, q' : Q}$ ▷ Solve for reachability matrix.
3:     $M : \mathbb{B}^{|Q| \times |Q| \times |V|} \leftarrow \big[\blacksquare \text{ if } \exists s : \Sigma \mid (v \to s) \in P \wedge (q \overset{\varphi}{\to} q') \in \delta \wedge \varphi(s) \text{ else } \square \big]_{q, q' : Q, v : V}$
4:     **for** i **in** $\big[0, \lceil \log_2(|Q||V|) \rceil \big]$ **do**   ▷ Solves matrix exponential, $\exp(M_0)$.
5:         DONE $\leftarrow \blacksquare$
6:         **for** $\langle p, r, w \rangle$ **in** $Q^2 \times V$ **do**   ▷ Iterates one step of $M_{i+1} = M_i + M_i^2$.
7:             **if** $M[p, r, w]$ **or not** $R[p, r]$ **then continue**
8:             $Q_{pr} \leftarrow \{q : Q \mid R[p, q] \wedge R[q, r]\}$ ▷ Consider reachable states between p and r.
9:             $M[p, r, w] \leftarrow \blacksquare \text{ if } \exists q : Q_{pr}, x, z : V \mid M[p, q, x] \wedge M[q, r, z] \wedge (w \to xz) \in P \text{ else } \square$
10:            **if** $M[p, r, w]$ **then** DONE $\leftarrow \square$
11:        **if** DONE **then break**
12:    **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 whether $M_\infty[q_\alpha, q_\omega, S] = \blacksquare$ for some $q_\omega : F$. Otherwise if no such q_ω 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: Same as `cfl_fixpt` (Alg. 2), $M_\mathbb{B}[q_\alpha, q_\omega, S] = \blacksquare$ for some $q_\omega : F$, and $M_\mathbb{B} = M_\mathbb{B} + M_\mathbb{B}^2$.

1: **procedure** `reg_build`$\left(M_\mathbb{B} : \mathbb{B}^{|Q| \times |Q| \times |V|}, \langle \Sigma, V, P, S \rangle : \text{CFG}, \langle Q, \Sigma, \delta, q_\alpha, F \rangle : \text{NFA}\right)$

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

3: $\quad M : \text{GRE}^{|Q| \times |Q| \times |V|} \leftarrow \left[\{s : \Sigma \mid M[q, q', v] \wedge (q \xrightarrow{\varphi} q') \in \delta \wedge (v \rightarrow s) \in P \wedge \varphi(s)\}\right]_{q, q' : Q, v : V}$

4: $\quad$ **for** i **in** $\left[0, \lceil \log_2(|Q||V|) \rceil\right]$ **do**

5: $\qquad M' \leftarrow M$

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

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

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

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

10: $\qquad$ **if** $M = M'$ **then break else** $M \leftarrow M'$

11: $\quad$ **return** $\bigvee_{q_\omega : F} M[q_\alpha, q_\omega, S]$ $\qquad$ ▷ 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, $c < |\sigma|, \forall \sigma : \mathcal{L}(e)$.

1: **procedure** `reg_dcode`$\left(e : \text{GRE}, C : \mathbb{N}^{|\Sigma|^{c \approx 4}}, k : \mathbb{N}\right)$

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

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

4: $\quad$ **repeat**

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

6: $\qquad \mathcal{E}' \leftarrow \left[\langle \sigma \cdot a, \partial_a e, p + \ln P(a \mid \sigma) \rangle \mid a \in \text{follow}(e)\right]$

7: $\qquad \mathcal{T} \leftarrow \mathcal{T} \mathbin{+\!\!+} \left[\langle \sigma, p \rangle \mid \langle \sigma, e, p \rangle \in \mathcal{E}' \wedge \varepsilon \in \mathcal{L}(e)\right]$

8: $\qquad \mathcal{E} \leftarrow \left[\langle \sigma, e, p \rangle \in (\mathcal{E} \mathbin{+\!\!+} \mathcal{E}') \text{ sorted by } p\right]$

9: $\quad$ **until** interrupted or $\mathcal{E}$ is empty.

10: $\quad$ **return** $[\sigma \mid \langle \sigma, p \rangle \in \mathcal{T}_{0..k} \text{ sorted by } p]$ $\qquad$ ▷ Skim off top-k repairs by c-gram 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

The foregoing architecture can be translated to series of high-performance GPU kernels. Our strategy will be to maximize GPU utilization by distributing the workload for each stage across as many independent threads as we can simultaneously dispatch. Each thread will be responsible for writing to a dedicated portion of a shared buffer without locking or external 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 $b_1 : \mathbb{N}^{d_1}, \dots b_n : \mathbb{N}^{d_n}$ containing encoded data, does some arithmetic, and returns a single output buffer, $b_{\text{out}} : \mathbb{N}^d$. On a GPU, all memory must be sized ahead of time, as dynamic allocation is forbidden during a GPU kernel's execution. The main challenge of GPU programming then, becomes careful memory management and efficiently mapping aggregate datatypes to and from the integers. Conceptually, each $\langle r, c, v \rangle$ triple will be dispatched to a single GPU thread with global read access to the input buffers and exclusive write access to a contiguous region of the output buffer. Consistent with the PRAM model used in Theorem 3.1, each thread will correspond to a single processor, with $|Q|^2 |V|$ threads in total. Absent a GPU, this can be rewritten as a triply-nested loop, subject to additional latency.

For the 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 characteristic functions $\texttt{nt\_enc} : \Sigma \to 2^V$, $\texttt{nt\_dec} : V \to 2^\Sigma$ for nonterminal encoding and decoding, and index sets $\Sigma \leftrightarrow \mathbb{N}, V \leftrightarrow \mathbb{N}, Q \leftrightarrow \mathbb{N}$ for getting in and out of the $\texttt{uint}$ domain, with $|V|, |Q| \lesssim 10^3$ adjustable upward if memory permits.

The parse chart M can be represented as a bit-packed integer matrix $\texttt{uint32}^{|Q| \times |Q| \times |V|}$, whose layout testifies to four properties of each $\langle q, q', v \rangle$ triple: (1) the first bit encodes the dis/equality predicate φ, (2) the next 25 bits designate terminal participation (if $\exists s : \Sigma. \varphi(s) \land (q \xrightarrow{\varphi} q') \in \delta$), (3) the next five bits memoize the minimum $i_{\min}$ such that $M_{i_{\min}}[q, q', v] \& 1 = \blacksquare$ for short-circuiting (see Line #7 of Alg. 3), and (4) the lowest-order bit denotes parsability, i.e., $q \rightsquigarrow q' \vdash v$. Note the decoder must acknowledge the possibility that v can simultaneously parse (a) an arc $q \to q'$ and (b) a path $q \rightsquigarrow q'$, so each branch can be explored. This is depicted below in little-endian format:

$$\left[\blacksquare, \underbrace{\square, \square, \dots, \blacksquare}_{s:\, \Sigma \Leftrightarrow \mathbb{B}^{25}}, \square, \blacksquare, \underbrace{\square, \square, \square, \square, \blacksquare}_{i_{\min}:\, \mathbb{N}_{\leq 32} \Leftrightarrow \mathbb{B}^5}, \underset{v:\, V}{\blacksquare} \right] : \texttt{uint32}$$

Once $\texttt{cfl\_fixpt}$ (Alg. 2) is complete, we can calculate the total amount of memory needed to allocate $G_\cap$ by counting constituents in the parse chart. Being an algebraic datatype, the GRE can be flattened according to a variety of allocation models. We will use the following memory layout,

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

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

In total, the GPU should have at least 4 GBs of onboard memory to accommodate language intersections with up to 10^3 states and nonterminals ($|Q|^2 \times |V| \lesssim 10^6 \times 10^3 \times 32$ bits $\approx$ 4 GBs), however occupancy can be roughly halved by exploiting the upper-triangular structure of M.

6.2 Training the reranker

After decoding, we have a list of repair candidates that are all valid, nearby and at least somewhat plausible, however it is possible this list may be quite long. No reasonable user would be expected to skim through more than a few dozen candidates to select their intended repair, especially since they could have presumably written it themselves in a few seconds. So, we will proceed to rerank the list. If we can guarantee the candidate repairs are sufficiently exhaustive, they should include with high probability the true repair, which then need only be surfaced into the user's field of view.

The ensuing method falls under the umbrella of the *learning-to-rank* (LTR) problem in machine learning – using their terminology, the broken code snippet would be called a *query* and the list of repairs, *documents*. To discuss the reranker, we must now overload some concepts, so the reader is trusted to contextually interpret $\mathcal{L}$ as denoting a *loss* instead of a language, and the derivative [4] as the directional rate of change of a differentiable manifold over the parameter space of a neural network. Matrix multiplication remains more or less the same, except now over the reals.

The reranker employs a transformer-encoder architecture to map both the query (broken code snippet, denoted $\underset{\sim}{\sigma} : \bar{\ell}$) and the document (candidate repair, denoted $\sigma : \ell_\cap$) to a p-dimensional real vector space, $\mathbb{R}^p$. We elide the definition of a transformer-encoder (see Strobl et al.'s survey [50]), except to say that it is a function, E_θ, which takes a string and a positional encoding, and returns an embedding, $E_\theta : (\Sigma \times \mathbb{N})^n \to \mathbb{R}^p$ where θ are learnable parameters. We introduce a second encoder, $E'_\theta : (\Sigma \times \mathbb{N} \times \mathbb{Z}_4)^{[m,n]} \to \mathbb{R}^p$, which takes in a Levenshtein alignment (LA $: \Sigma^n \times \Sigma^m \to \mathbb{Z}_4^{[m,n]}$) tracking edit locations and types for each query-document pair. Finally, a multilayer perceptron (MLP$_\theta : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^+$) processes the embedding to produce a scalar relevance score:

$$f_\theta(\underset{\sim}{\sigma}, \sigma) : \Sigma^n \times \Sigma^m \to \mathbb{R}^+ = \text{MLP}_\theta\Big(E_\theta\big(\underset{\sim}{\sigma}, [i]_{i\in[0,n)}\big), E'_\theta\big(\sigma, [i]_{i\in[0,m)}, \text{LA}(\underset{\sim}{\sigma}, \sigma)\big)\Big) \tag{19}$$

Our training objective will be to minimize the tempered *softmax* or listwise cross-entropy loss,

$$\mathcal{L}(\theta) = -\sum_{q\in\mathcal{Q}} \log\left(\frac{\exp\big(f_\theta(q, d^*)\tau^{-1}\big)}{\sum_{d\in\mathcal{D}_q}\exp\big(f_\theta(q, d)\tau^{-1}\big)}\right) \tag{20}$$

where $\mathcal{Q}$ is the set of training queries, $\mathcal{D}_q$ is the set of candidate repairs for query q, and $d^* \in \mathcal{D}_q$ is true repair. The temperature parameter, τ controls the sharpness of the softmax distribution, encouraging parameter settings that result in the true repair being assigned higher priority – the closer to zero, the greater the loss will be for underestimating the relevance of the true repair.

More concretely, we depict a single instance of the training data in Fig. 11. The reranking model sees a (1) query, (2) document, (3) positional encoding and (4) Levenshtein alignment, and returns a numerical score. Once the relevance scores are obtained, we calculate the cross-entropy loss across the top-k scoring documents and backpropagate. In practice, this update is averaged across a batch $\langle q_i, [d_j]_{0...k}\rangle_{i=0...|B|^{\approx 16}}$ of repair instances to reduce noise. The batch update rule is a standard variant of stochastic gradient descent $\big(\theta' \leftarrow \theta - \alpha\nabla_\theta\mathcal{L}(\theta)\big)$ with momentum (AdamW), where the learning rate is in the range $\alpha \approx 10^{-4}$. A more exhaustive description of the architectural details and hyperparameter settings used for training the reranker can be found in Appendix C.

q:	NAME	=	NAME	)	NAME	
PE:	0	1	2	3	4	
d_1:	NAME	=	NAME	(	NAME	)
PE:	0	1	2	3	4	5
LA:	0	0	0	2	0	1
d_2:	NAME	(	NAME	)	NAME	
PE:	0	1	2	3	4	
LA:	0	2	0	0	3	
d_3:	NAME	=	NAME	.	NAME	(	)
PE:	0	1	2	3	4	5	6
LA:	0	0	0	2	0	1	1

Fig. 11. Transformer data encoding.

---

[4] There is a connection to Brzozowski's derivative, but to refrain from digression here we refer the reader to [21] for details.

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., decoding, reranking, 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 precision and latency.

7.1 Experimental setup

In the following set of experiments, we use syntax errors and fixes from the Python language. Python code snippets are abstracted as a sequence of lexical tokens using the official Python 3.8.11 parser, erasing alphanumeric identifiers and literals but retaining all other keywords. Accuracy is evaluated across a test set of pairwise errors and repairs by checking for lexical equivalence with the ground-truth repair, following the same methodology as Sakkas et al. (2022) [46].

We use the Precision@k statistic, which measures the frequency of the true repair appearing in the top-k results, across a dataset of repair instances. Specifically, given a repair model, $R : \Sigma^* \to (\Sigma^*)^t$ and a test set, $\mathcal{D}_{\text{test}}$, containing pairwise aligned errors ($\underset{\sim}{\sigma}$) and fixes (σ'), we define Precision@k as:

$$\text{Precision@k}(R) = |\mathcal{D}_{\text{test}}|^{-1} \sum_{\mathcal{D}_{\text{test}}} \mathbb{1} \left[\sigma' \in R(\underset{\sim}{\sigma})_{0\ldots k} \right] \tag{21}$$

Our full dataset [56] consists of 2×10^4 naturally-occurring pairs of Python errors and human fixes from StackOverflow, which we use to evaluate the precision of each model at blind recovery of the ground truth repair. From the StackOverflow dataset, we filter for syntax errors shorter than 80 tokens and fewer than four lexical edits apart from the corresponding repair, then divide the remaining repairs into two disjoint sets: a training set ($\mathcal{D}_{\text{train}}$) of 4,586 repair instances and a test set ($\mathcal{D}_{\text{test}}$) of 2,238 repair instances, balanced across each length interval and edit distance, i.e., $\mathcal{D}_{\text{test}} = \{ \langle \underset{\sim}{\sigma}, \sigma' \rangle \mid \lfloor |\underset{\sim}{\sigma}|/10 \rfloor \in [0, 8], \Delta(\underset{\sim}{\sigma}, \sigma') \in [1, 3] \}$, each test bin containing at least 50 instances.

To train the reranker, we augment each instance in the training and test set with a list of repair confounders. Each instance consists of a tuple, $\langle \underset{\sim}{\sigma} : \bar{\ell}, \sigma' : \ell_\cap, \boldsymbol{\sigma} : \ell_\cap^{\leq 10^3} \rangle$, with a single syntax error ($\underset{\sim}{\sigma}$), the ground truth repair, (σ'), and up to 10^3 confounders ($\boldsymbol{\sigma}$) sampled without replacement from $\ell_\cap$ using our pretrained 4-gram model. We then train the reranker on $\mathcal{D}_{\text{train}}$ for 13,000 steps which takes ~ 4 hours, and evaluate on $\mathcal{D}_{\text{test}}$. Hyperparameters are provided in Appendix C.

For our final repair procedure, we use the CNF Python grammar, G_{Python} and let d_{max} be the smallest value such that $\ell_\cap = \mathcal{L}(G) \cap \mathcal{L}(L(\underset{\sim}{\sigma}, d_{\text{max}} - 1))$ is nonempty. We decode $\ell_\cap$ with the same pretrained 4-gram model used in reranker training, and pass the top 10^3 results by 4-gram probability to the transformer encoder, then finally rerank the top 10^3 by softmax probability and measure the Precision@k across repairs of differing length and edit distance in the test set.

We compare our method with two external baselines, Seq2Parse and Break-It-Fix-It (BIFI) [58], on the same test set. 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`. [5] For BIFI, we use the Round 2 breaker and fixer from commit `ee2a68c`,[6] 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-$\{1, 2 \times 10^4\}$ repairs. Finally, for Tidyparse, we use a standard Apple MacBook M4 Max with 128 GB of memory.

---

[5]https://github.com/gsakkas/seq2parse/tree/7ae0681f1139cb873868727f035c1b7a369c3eb9

[6]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 [56]. We preprocess the dataset to lexicalize all code snippets, then filter by length and distance shorter than 80 lexical tokens and under four edits, i.e., with Levenshtein distance under four lexical edits $\left(|\Sigma| = 88, |\sigma| < 80, \Delta(\sigma, \sigma') < 4\right)$. We depict the length, edit distance, normalized edit locations and stability profile in Fig. 12.
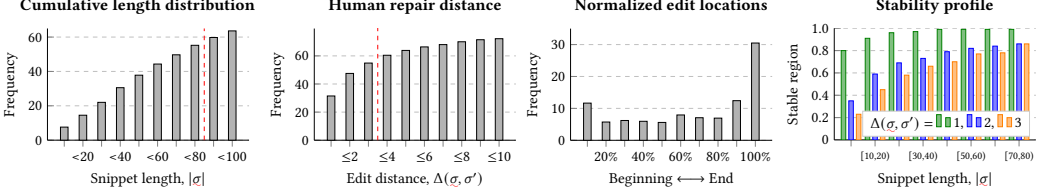
Fig. 12. Repair statistics across the StackOverflow dataset, of which Tidyparse can handle about half in under ~3s and ~4 GB. Larger repairs and edit distances are possible, albeit requiring additional time and memory.

We observe that slightly over 6,700 code snippet pairs in the StackOverflow dataset contain fewer than 80 tokens and four lexical edits, which are computationally feasible to process in a few hundred milliseconds. We also note a slight primacy or recency bias in the edit locations, evidenced by a large fraction of human repairs which modify the boundaries of the broken code snippet.

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.

For an intuition about the size of the language intersections involved in syntax repair, volumetric analysis will be helpful, particularly in understanding the influence of snippet length and edit distance on language intersection volume. To measure the intersection volume we will form the $L(\sigma, \Delta(\sigma, \sigma'))$ automaton, intersect it with the Python grammar, then automatize the resulting regular expression and finally compute the DFA transfer matrix using the method described in § 5.4 to obtain the exact volume. For a given (error, fix) pair, this tells us how many repairs of equal or lesser distance exist in the Python language. Plotting intersection volume across the full dataset (Fig. 13), we observe a strong positive correlation with the Levenshtein margin and a mild correlation with snippet length. Fully materializing

Fig. 13. Language volume versus snippet length and edit distance for Python repairs.

$\ell_\cap$ is typically only feasible if we extend the Levenshtein radius up to one edit beyond the language edit distance (LED) $\left(\text{i.e., } d_{\max} \leq \text{LED}(\sigma)^7 + 1\right)$ after which it grows too large to exhaustively generate and must be sampled. Across all snippets where $\Delta(\sigma, \sigma') < 4$, approximately 54% matched LED(σ), 35% had an edit distance of LED(σ) + 1 and 11% had a distance of LED(σ) + 2.

---

[7]Where LED(σ) is shorthand for LED(σ, ℓ) = min $\left\{d_{\max} : \mathbb{N} \mid \mathcal{L}\left(L(\sigma, d_{\max})\right) \cap \ell \neq \varnothing\right\}$ with ℓ being the Python language.

7.3 StackOverflow evaluation

For our first experiment, we measure the top-1 precision of our repair procedure at various lengths and Levenshtein distances, comparing Tidyparse against Seq2Parse, and BIFI on the same test set. Each bin in the test set contains at least 50 distinct repairs sampled uniformly at random from the StackOverflow dataset, none of which were present in the training set of any repair model.
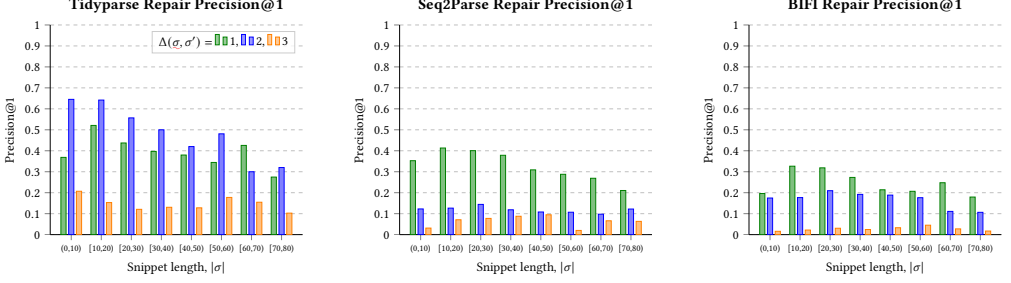

Fig. 14. Probability of the first recommendation matching the true repair for Tidyparse, Seq2Parse and BIFI repair precision at various lengths and Levenshtein distances.

Tidyparse attains state-of-the-art top-1 repair precision versus both models by a wide margin. Unexpectedly, precision does not monotonically decrease with edit distance, as Tidyparse's double-edit Precision@1 slightly outperforms single-edit Precision@1 across the test set. Although Seq2Parse outperforms BIFI by a lower margin, results are also mixed for the 2-edit repair case. Similar to Fig. 13, the nonlinear correlation between edit distance and repair precision holds across all three models, as does a slightly negative correlation between repair length and precision.

For the next experiment, we evaluate the BIFI model, giving it a generous compute and latency advantage with an unlimited time budget to sample 2×10^4 repairs, and compare the Precision@10 of our approach with a 10s timeout. As Tidyparse uses a 4-gram model for decoding, it can sample a much larger candidate set in the time allotted, but must use a transformer-based reranker after decoding to sort the top-10^3 repairs. Since the Seq2Parse reference implementation does not support sampling more than one repair, we do not compare its Precision@k for higher k values. The raw data from these experiments can be found in Appendix E.
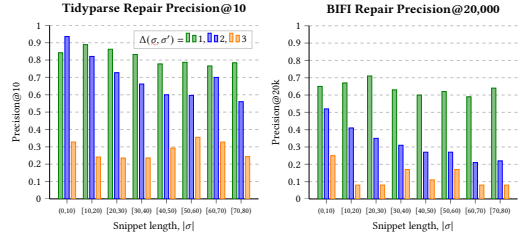

Fig. 15. Probability of the true repair being in the first ten Tidyparse repairs, and the first 2×10^4 BIFI repairs.
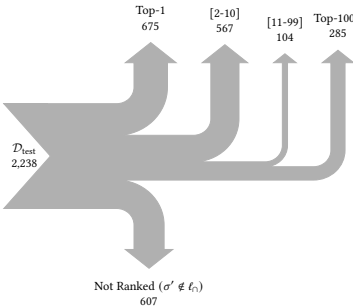

Fig. 16. Outcomes in the repair pipeline.

We present a Sankey diagram of the Tidyparse repair pipeline in Fig. 16. Across 2,238 test set repairs filtered by length and distance ($\lfloor |\sigma|/10 \rfloor \in [0, 8], \Delta(\sigma, \sigma') < 4$), we evaluated Tidyparse with a timeout of 10s and tracked individual repair outcomes. In 607 cases, the true repair was not contained in the language intersection and thus never sampled, in 1,631 cases the true repair was sampled, of which 675 cases the first prediction matched the true repair, in 1,242 cases, the true repair was in the top-10 results, and in the remaining 389 cases the true repair was drawn, but ranked lower than 10th in the final results.

7.4 Internal evaluation

The primary question of interest here is, to what extent does the neural reranker improve precision relative to a naïve decoding strategy? For comparison, we use an 4-gram based repair sans reranking. That is, we decode the language intersection with a 4-gram model, sort the repairs by their respective 4-gram probabilities, and without further processing, evaluate Precision@$10^{\{0,1,2,3\}}$.
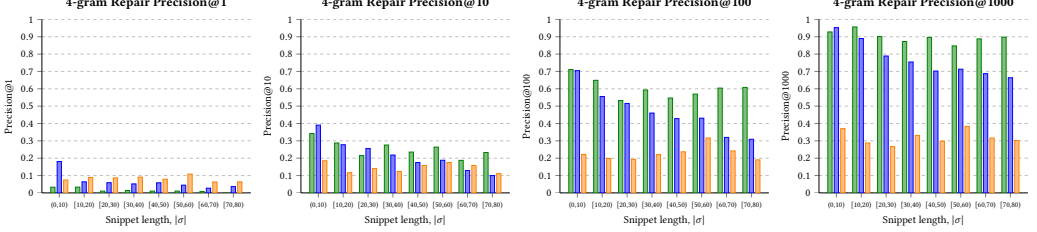

Fig. 17. 4-gram repairs. 4-gram Precision@1000 is an upper bound on Tidyparse Precision@k, since the latter only reranks the top-10^3 most probable 4-gram sampled repairs from the language intersection.

We can quantify the ranking improvement by comparing CDFs of the true repair's rank across the test set of human repairs, before and after reranking the top-10^3 sampled repairs (Fig. 18). Since we decode but do not consider less probable repairs, reranking does not affect repairs originally ranked lower. Repairs initially ranked in the top-10^3 results by 4-gram probability tend to place between 1st and 10th in about 75% of instances after reranking. It is possible the true repair can be ranked higher before reranking than after, which occurs in $\sim 4\%$ of cases.

Finally, we investigate the impact of increased parallelism on repair throughput. To simulate a realistic editing scenario, we measure the end-to-end wallclock runtime required to construct the Levenshtein automaton, form the intersection regex ($S_\cap$), decode and rank the entire intersection language ($\ell_\cap$). Then, we swap in our GPU implementation of the algorithm described in § 6 as a replacement for the CPU version and compare the individual repair timings across the same test set.
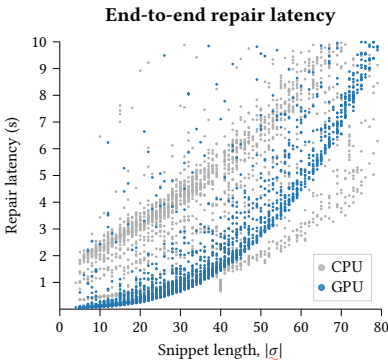

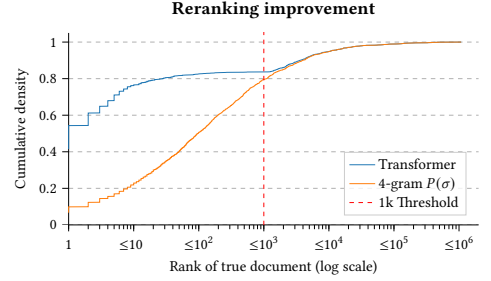Fig. 18. Observed improvement in repair rank with and without the transformer reranker.

Fig. 19. End-to-end repair timings.

As shown in Fig. 19, latency depends on various factors but supports the complexity analysis (§ 3), exhibiting a clearly superlinear but subexponential runtime profile. While real-world performance can vary based on LED, intersection volume and other load factors, the GPU runtime generally has lower variance and confers a 2-3x speedup across most common repair scenarios. Our GPU implementation is able to exhaustively decode the intersection for almost all instances before 10s, however the equivalent CPU version may struggle to meet the same latency target, especially on longer or multi-edit repairs. While the 10s timeout can be arbitrarily extended, we anticipate a much longer delay would begin to tax the patience of most users, and therefore consider it a reasonable upper bound for repair latency.

8 DISCUSSION

The main lesson we draw from § 7 is that it is feasible to significantly improve the precision of real-time syntax repair by incorporating syntactic constraints such as edit distance, then sampling and evaluating a large set of candidate repairs using a fast primary decoder and a more intensive secondary reranker. Though sample-efficient, transformer decoding comes at considerable cost to throughput, resulting in fewer repairs being discovered in a fixed amount of time.

Our approach uses a grammar and a high-throughput c-gram decoder to fetch the initial candidate repairs, then employs the transformer-encoder to rerank only the top-scoring repairs from the retrieved set. This allows us to repair errors in real-world programming languages and provides far more flexibility and controllability during the repair process, resulting in significantly higher precision on downstream repairs and ultimately, a smoother user experience.

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 single-shot neural repair. This is especially true for small-radii Levenshtein balls, where the intersection language is small enough to be completely enumerated and ranked. For larger radii, we can still achieve state-of-the-art precision by sampling a representative subset within a fixed timeout.

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@10 of our method is competitive with BIFI's Precision@2×10^4, while requiring only a fraction of the inference time. Unlike neural syntax repair models, Tidyparse can sample directly from the language specification, removing the possibility of hallucination. The emphasis on completeness is especially useful for discovering small or contextually improbable repairs, which are easily 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 reranker, if the correct repair is never retrieved, it will be for naught. By maximizing the total number of unique valid repairs, we increase the probability of retrieving natural repairs to give the reranker the best chance of surfacing them to the user.

One might be tempted to model syntax repair as a rejection sampling problem, but as Fig. 13 portrays, this strategy would be mistaken. Even if checking a single repair for validity takes just 1 ms, complete enumeration could take 24+ hours, and we have mere seconds at most. While rejection sampling has lower latency to find admissible repairs, it wastes a tremendous amount of computation and scales poorly with edit distance. It is far better to spend more computation upfront by performing the intersection in a way that avoids rejection and returns natural repairs.

Likewise, methods that rely on decoding large language models appear to face a similar dilemma. As shown in Fig. 15, even if we sample thousands of repairs from BIFI, an LLM specifically trained on syntax repair, it is possible to miss natural valid repairs of a given distance that would be easily found by an extensive c-gram search of $\ell_\cap$, plus reranking. This suggests completeness may be equally, if not more important, than sample efficiency for the purposes of evaluating candidate repairs. Indeed, if we compare the 4-gram Precision@100 in Fig. 17 with BIFI's Precision@2×10^4, 4-gram Precision@100 is highly competitive even without any post-decoder reranking.

Taken together, these results provide strong evidence to support the central claim made in the introduction (§ 1): *existing syntax repair methods simply generate far too few repairs to be effective.* By extensively generating and evaluating a large quantity of repairs within a fixed edit distance, we show it is possible to predict the author's intent far more reliably, with greater precision and lower latency than competing methods which rely solely on transformer-based neural networks.

8.1 Limitations and future work

We identify three broad categories of limitations in Tidyparse and suggest directions for future work: naturalness, complexity, and toolchain integration.

8.1.1 Naturalness. Firstly, Tidyparse does not currently support intersections between weighted CFGs and weighted finite automata, a la Pasti et al. [42]. This feature would allow us to put transition probabilities on the Levenshtein automaton corresponding to edit probability, then construct a weighted intersection grammar. With this, one could preemptively discard unlikely productions from $G_\cap$ to reduce the complexity of intersection in exchange for relaxed completeness. We also hope to explore alternate sampling strategies such as sequential Monte-Carlo [35] and denoising diffusion models [6] with an inductive bias for structured sampling via Levenshtein edits.

The reranker is currently evaluated over lexical tokens, but we expect that a more precise ranking function could be constructed by incorporating names and numbers from the original source code and then scoring plaintext. Furthermore, the decoder only considers each candidate repair $P_\theta(\sigma')$ in isolation, returning the most probable candidates independent of the original error. This could be improved by conditioning the decoding on the broken sequence (σ), parser error message (m), original source (s), and possibly other contextual priors to assist sample efficiency.

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 various encoding tricks and a more efficient GPU implementation, 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 and using an efficient representation, are critical factors in scaling up our technique.

8.1.3 Toolchain integration. 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. One solution would be to just use the line surrounding the caret position, however a more complete solution requires careful editor 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, and must be filtered by some form of compiler or incremental type checker before presenting them to the user. 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.

Extending language intersections to handle type error repair requires leaving the domain of syntax and entering the much more daunting world of semantics – here one must contend with difficult questions in mathematical logic and finite model theory. One direction would be to collapse these problems down to automata theory using MSO over words via the Büchi-Elgot-Trakhtenbrot theorem. Another direction would be to increase the expressivity of the grammar, using something like conjunctive grammars [41]. A third approach would be to adopt the framework of contextual modal type theory, then study the behavior of Levenshtein edit distance on modal accessibility in weak substructural type systems like the Lambek calculus [43]. We leave this for future work.

9 RELATED WORK

Our work draws a threefold correspondence between well-known techniques in (1) formal language theory, (2) program analysis and (3) incremental decoding. We will first survey these topics, then turn our attention to machine learning, with which we compare and partly use for reranking.

9.1 Formal language theory

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) [45], Earley [20] and others. These parsers have roughly cubic complexity with respect to the length of the input string.

As shown by Valiant [52], Lee [34] 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 applicability.

Bar-Hillel [7] proves the closure of CFLs under intersection with regular languages, but does not elaborate on how to construct the corresponding grammar. Salomaa [47] and Pasti et al. [42] provide helpful insights into constructing the intersection grammar, and Nederhof and Satta [40] specifically consider finite CFL intersections, but seem unaware of the connection to CFL reachability. Our work specializes Bar-Hillel intersections to Levenshtein automata in particular, and more generally acyclic automata using a refinement of Salomaa's construction [47] based on CFL reachability.

9.2 CFL reachability

Our contribution is closely related to the literature on CFL reachability. In brief, the CFL reachability problem seeks to determine, given an edge-labeled graph and distinguished vertex pair, $\langle v, v' \rangle$, whether there is a path, $v \leadsto v'$, whose concatenated edge labels are contained in the CFL. For a deeper overview, see Zhang and Su [59]. This problem has been known [44] for some time [32] to have broad applications to program analysis and as our work finds, to syntactic program repair.

From a complexity-theoretic perspective, the CFL reachability problem is known to be at worst subcubic [14] with polylogarithmic time factors. Koutrus and Deep [33] present a fine-grained complexity analysis, with concurrent work by Istomina et al. [31] expanding on fine-grained reductions. Muravev and Grigorev explore how to accelerate this technique on a GPU [39].

Surprisingly absent from the literature on CFL reachability is a discussion of the Bar-Hillel construction, regular expressions for witnessability, or the use of Brzozowski's derivative for incremental decoding. Nor does the literature specifically consider the parallel complexity of intersection nonemptiness between CFLs and acyclic automata such as the Levenshtein NFA (§ 4.3). In Theorem 3.1 we give a constructive proof of finite intersection nonemptiness, borrowing the matrix multiplication technique from CFL reachability to build a star-free regular expression that we decode using the Brzozowski derivative. This technique sheds new light on the Bar-Hillel construction, and translates to a simple and efficient implementation which is fully compatible with left-to-right incremental decoding techniques used in probabilistic language modeling.

9.3 Language equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [24] 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's and Newton's. First studied by Brzozowski [9, 10] and Antimirov [5], 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, 38] and regular expression matching [49, 53].

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 [38], to our knowledge, they were never specifically considered for code completion or syntax error correction.

9.4 Syntax repair

In finite languages, syntax repair corresponds to spelling correction, a more restrictive and largely solved problem. Schulz and Stoyan [48] 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 string is syntactically 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 [30] 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 parsers 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 probability Levenshtein-CFL reachability, which seeks to find the most natural syntactically valid repair within a fixed Levenshtein distance.

Diekmann and Tratt [17] present a rule-based syntax repair tool that retrieves the complete set of minimum-cost repairs, but only works for deterministic CFLs, a proper subset of the CFL family which admit a linear-time parser. Their cost model is based on insertion and deletion, and does not consider probability or non-minimal edit distance. Tidyparse can handle arbitrary CFLs and generate repairs within an arbitrary edit distance, using a Levenshtein cost model.

Zhang et al. [60] introduce OrdinalFix, which uses CFL reachability to repair compiler errors, however their method only returns admissible repairs and not necessarily probable ones. As they do not consider the problem of maximum-probability repairs, nor use any form of ranking to sort the results by naturalness or probability, we do not compare with their work.

9.5 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 to find the maximum probability samples. A classic example is Viterbi decoding, which is used to find the most likely path through a hidden Markov model (HMM), a kind of weighted automaton.

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 probability. 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 CFGs is often intractable, but we can approximate it using a beam search over c-grams, then rerank top-scoring results.

A popular solution to k-best decoding in the NLP literature is a technique called cube-pruning [11, 29], which samples maximum probability paths through a hypergraph. We take inspiration from this technique and adapt it to the setting of constrained decoding from finite CFGs.

An alternate line of work originates from combinatorics [28] and Boltzmann sampling [19], which constructs a generating function for the language and samples it uniformly. This technique has applications to constraint satisfaction and model counting problems in formal languages.

A third approach would be to use some form of constrained decoding [36, 51, 55] such as sequential Monte Carlo to steer an autoregressive LLM, as proposed by Lew et al. [35]. These techniques show promise for program repair, however, the question of whether to use left-to-right decoding or some other strategy is still unresolved in the language modeling community. For example, there is an emerging class of flow-based or structured denoising diffusion models [6] which starts from a noise distribution and iteratively denoises it by sampling one or more edits at random locations with each decoding step. Typical work focuses on audiovisual data, but very recent work by Havasi et al. [27] adapt this to the Levenshtein edit model for generating source code. Although these models do not yet use CFGs or consider language intersections, they are inherently more fault-tolerant than decoders which require expensive backtracking-style search.

9.6 Learning-based program repair methods

The last decade has seen a surge of progress in programming with large language models. That work is primarily based on methods from differential calculus and continuous optimization, leading to the so-called *naturalness hypothesis* [3], which suggests programming languages are not so different from natural ones. In contrast, PL theory takes the view that languages are essentially discrete sets governed by logical calculi. Programming, thus viewed, is more like a mathematical exercise in constraint satisfaction. These two approaches have more in common than would seem.

From an applied perspective, a number of gradient-based methods have been introduced to repair programming errors [4, 13, 18]. 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.

Two prior works specifically address syntax repair, Break-It-Fix-It (BIFI) [58] and Seq2Parse [46]. 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 generalizes 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.

Recent work by Merrill et al. [37] and Chiang et al. [12] suggest that the issue with generalization may be more foundational: transformer-based language models, a popular class of neural language models used in program synthesis and repair, are fundamentally less expressive than context-free grammars, which formally describe the syntax of many programming languages. This suggests such models, despite their useful approximation properties, are ill-suited for the task of end-to-end syntax repair. Yet, as our work demonstrates, they can be useful for resolving ambiguity between valid repairs of differing probability or reranking a set of repair candidates drawn from a CFL.

Using the RASP model from Weiss et al. [54], Yang et al. [57] characterize the expressive power of hard-attention transformers in terms of star-free regular expressions. While their work uses formal language theory to investigate language learnability and structural priors in transformers, it shares fruitful connections to CFL reachability, which similar to RASP, treats matrix multiplication as a kind of programmable interface into which various state tracking problems and static analysis tasks can be compiled and analyzed in a common linear algebraic framework.

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.

Various codesign patterns have emerged to fuse the naturalness of neural language models with the precision and completeness 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 steer language models to search for valid programs via a reinforcement learning or hybrid neurosymbolic approach. However, these approaches can introduce significant latency into the repair process and lack strong statistical guarantees.

Our work takes 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 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 frees up our limited compute budget for more efficient search and ranking at inference time.

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 probable repairs receive the highest priority.

We have implemented our approach and demonstrated its viability as a tool for syntax assistance in real-world programming languages. Tidyparse [8] is capable of generating repairs for invalid source code in a range of practical languages, in addition to Python. We plan to continue expanding the prototype's autocorrection functionality to cover an even broader range of real-world programming languages. We envision a few primary use cases for it: (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.

10.1 Acknowledgements

---

[8]An artifact for Tidyparse is currently available as a browser application, supporting single-line syntax repairs in Python and other CFLs: 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, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) 51, 4 (2018), 1–37. https://arxiv.org/pdf/1709.06182.pdf

[4] 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

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

[6] Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. 2021. Structured denoising diffusion models in discrete state-spaces. Advances in neural information processing systems 34 (2021), 17981–17993.

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

[14] Dmitry Chistikov, Rupak Majumdar, and Philipp Schepper. 2022. Subcubic certificates for CFL reachability. Proceedings of the ACM on Programming Languages 6, POPL (2022), 1–29.

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

[16] Eliezer Dekel, David Nassimi, and Sartaj Sahni. 1981. Parallel matrix and graph algorithms. SIAM Journal on computing 10, 4 (1981), 657–675.

[17] Lukas Diekmann and Laurence Tratt. 2018. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. arXiv preprint arXiv:1804.07133 (2018).

[18] 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.

[19] 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.

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

[21] Conal Elliott. 2019. Generalized Convolution and Efficient Language Recognition (Extended version). (2019).

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

[23] Philippe Flajolet. 2009. Analytic Combinatorics. Cambridge University Press.

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

[25] Joshua Goodman. 1999. Semiring parsing. Computational Linguistics 25, 4 (1999), 573–606.

[26] Dick Grune and Ceriel J. H. Jacobs. 2008. Parsing as Intersection. Springer New York, New York, NY, 425–442.

[27] Marton Havasi, Brian Karrer, Itai Gat, and Ricky TQ Chen. 2025. Edit Flows: Flow Matching with Edit Operations. arXiv preprint arXiv:2506.09018 (2025).

[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] E. T. Irons. 1963. An Error-Correcting Parse Algorithm. Commun. ACM 6, 11 (nov 1963), 669–673.

[31] Aleksandra Istomina, Semyon Grigorev, and Ekaterina Shemetova. 2023. Fine-grained reductions around CFL-reachability. arXiv preprint arXiv:2306.15967 (2023).

[32] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. ACM Sigplan Notices 39, 6 (2004), 207–218.

[33] Paraschos Koutris and Shaleen Deep. 2023. The fine-grained complexity of CFL reachability. Proceedings of the ACM on Programming Languages 7, POPL (2023), 1713–1739.

[34] 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

[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] João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, et al. 2025. Syntactic and semantic control of large language models via sequential monte carlo. arXiv preprint arXiv:2504.13139 (2025).

[37] 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.

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

[39] Ilia Muravev and Semyon Grigorev. 2025. Universal High-Performance CFL-Reachability via Matrix Multiplication (SOAP '25). Association for Computing Machinery, New York, NY, USA, 28–35.

[40] 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.

[41] Alexander Okhotin. 2001. Conjunctive grammars. Journal of Automata, Languages and Combinatorics (2001).

[42] 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

[43] Tikhon Pshenitsyn. 2025. First-Order ILL and Hypergraph Languages. arXiv preprint arXiv:2502.05816 (2025).

[44] Thomas Reps. 1998. Program analysis via graph reachability. Information and software technology (1998).

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

[46] 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.

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

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

[49] 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.

[50] Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. 2024. What formal languages can transformers express? a survey. Transactions of the Association for Computational Linguistics 12 (2024), 543–561.

[51] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving LLM code generation with grammar augmentation. arXiv preprint arXiv:2403.01632 (2024).

[52] 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

[53] 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.

[54] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers. In International Conference on Machine Learning. PMLR, 11080–11090.

[55] Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for LLMs. arXiv preprint arXiv:2307.09702 (2023).

[56] 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.

[57] Andy Yang, David Chiang, and Dana Angluin. 2024. Masked hard-attention transformers recognize exactly the star-free languages. Advances in Neural Information Processing Systems 37 (2024), 10202–10235.

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

[59] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. 344–358.

[60] Wenjie Zhang, Guancheng Wang, Junjie Chen, Yingfei Xiong, Yong Liu, and Lu Zhang. 2023. OrdinalFix: Fixing Compilation Errors via Shortest-Path CFL Reachability. arXiv preprint arXiv:2309.06771 (2023).

A   LEVENSHTEIN TOPOLOGY AND MATRICES

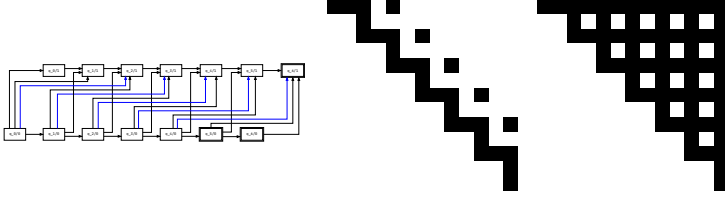These are useful for visually checking different implementations.

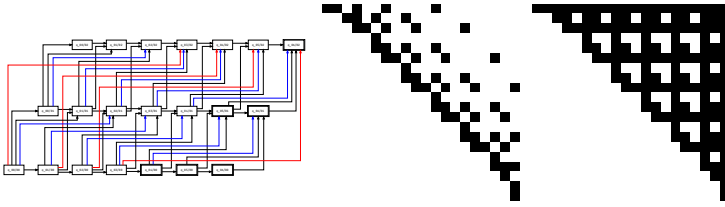Fig. 20. Lev($|\sigma|$=6, Δ=1) automaton, adjacency and reachability matrix.

Fig. 21. Lev($|\sigma|$=6, Δ=2) automaton, adjacency and reachability matrix.

Fig. 22. Lev($|\sigma|$=6, Δ=3) automaton, adjacency and reachability matrix.

Fig. 23. Lev($|\sigma|$=6, Δ=4) automaton, adjacency and reachability matrix.
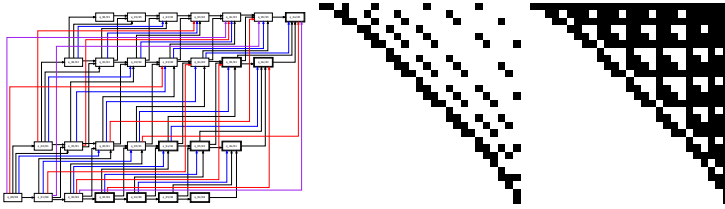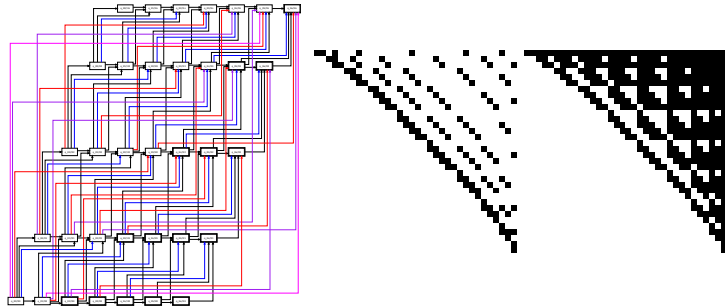
B LEVENSHTEIN AUTOMATA MINIMALITY

It is reasonable to ask whether the Levenshtein automaton defined in § 4.3 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 $\underset{\sim}{\sigma} : \bar{\ell} \cap \Sigma^n$ 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. $\varnothing = \mathcal{L}(\Sigma^p \cdot \underset{\sim}{\sigma}_{p\ldots n}) \cap \mathcal{L}(G)$ *implies the states* $[q_{i,d_{\max}}]_{i \in 0\ldots p}$ *are unnecessary.*

Therefore, 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 $\mathcal{L}(G_\cap)$. For example, let us consider the pruned L-NFA for the broken string $\underset{\sim}{\sigma} = $ [(+)] with $G = \{S \rightarrow (S) \mid [S] \mid S + S \mid 1\}$. Its longest parseable suffix and prefix are:

(B.1) _ _ +)] $\notin \mathcal{L}(G)$ ❌ $\wedge$ _ _ _)] $\in \mathcal{L}(G)$ ✅ $\implies [q_{n-i,0}]_{i \in 1\ldots s}$ are unnecessary.
(B.2) [(+ _ _ $\notin \mathcal{L}(G)$ ❌ $\wedge$ [(_ _ _ $\in \mathcal{L}(G)$ ✅ $\implies [q_{i,d_{\max}}]_{i \in 0\ldots p}$ are unnecessary.

Now we can prune the top leftmost and bottom rightmost states. Pictorially, this looks as follows:
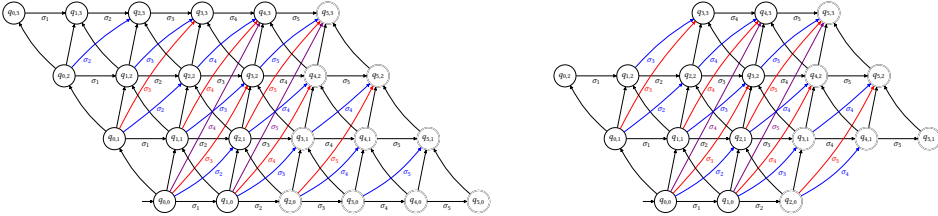

Fig. 24. Levenshtein NFA before and after prefix and suffix pruning.

C HYPERPARAMETER SETTINGS

Below is a listing of the hyperparameter settings used for training the reranking model:

- Input dimension: 100
- Encoder dimension: 512
- Attention heads: 4
- Encoder layers: 4

- Vocab size, $|\Sigma| = 94$
- Learning rate, $\alpha = 10^{-3}$
- Temperature, $\tau = 10^{-1}$
- Optimizer: AdamW

- Negative rate: 10^{-2}
- Batch size: 8
- Dropout: 10^{-1}
- Activation: GELU

The full parameters θ are partitioned into two sets, θ_e, θ_r, for the encoder and reranker layers. The encoder is pretrained on next-token prediction, then fine-tuned on the reranking task. During optimization, we use a smaller learning rate ($\alpha = 10^{-5}$) so as not to disturb the pretrained encoder parameters and a larger learning rate ($\alpha = 10^{-4}$) for the reranker parameters. We train the encoder for 2×10^4 steps and the reranker for 1.3×10^4 steps, taking ~ 4 hours on an Nvidia H100 GPU.

D 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
``` |

E 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 intersection or a 10 second timeout 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 20,000 samples. Each entry in the following table represents a pairwise disjoint subset of D_{test}, with at least 50 distinct Python syntax errors and repairs matching the length and distance criteria, sampled uniformly from the full StackOverflow dataset [56].

	Δ	Precision@1							
$|\sigma|$		$(0, 10)$	$[10, 20)$	$[20, 30)$	$[30, 40)$	$[40, 50)$	$[50, 60)$	$[60, 70)$	$[70, 80)$
Tidyparse	1	0.37	0.52	0.44	0.40	0.38	0.34	0.43	0.27
	2	0.65	0.64	0.56	0.50	0.42	0.48	0.30	0.32
	3	0.21	0.15	0.12	0.13	0.13	0.18	0.15	0.10
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

F SYMBOLS AT A GLANCE

Below we provide an inexhaustive listing of some common notation used throughout this paper.

Notation	Meaning
$G = \langle \Sigma, V, P, S \rangle$	CFG with terminals, Σ, nonterminals, V, productions, P, and start symbol S.
$A = \langle Q, \Sigma, \delta, q_\alpha, F \rangle$	Automaton with states, Q, transitions, δ, start state, q_α and final states, F.
σ	Syntactically invalid input string with a known target language.
$|\sigma|$	Length (number of terminals) of string, σ.
G^*	Chomsky Normal Form (CNF) grammar.
$G_\cap$	Intersection grammar formed by intersecting an automaton with a CFG.
$\ell_\cap, \mathcal{L}(G_\cap)$	Intersection language generated by some $G_\cap$.
$L(\sigma, k)$	Levenshtein automaton of radius k for a broken string, σ.
$[\ldots]$	Orange text is related to the symbolic predicate in the Levenshtein automaton.
$d_{\max}$	Maximum permitted Levenshtein edit distance (repair radius).
M	Matrix encoding the product construction $\mathcal{L}(G) \cap \mathcal{L}(L(\sigma, k))$.
$P@k$	Precision at rank k evaluation metric.