Syntax Repair as Language Intersection

ANONYMOUS AUTHOR(S)

We introduce a new technique for correcting syntax errors in arbitrary context-free languages. Our work comes from the observation that syntax errors with a small repair typically have very few unique small repairs, which can usually be enumerated within a small edit distance then ranked within a short amount of time. Furthermore, we place a heavy emphasis on precision: the enumerated set must contain every possible repair within a few edits and no invalid repairs. To do so, we reduce CFL recognition onto Boolean tensor completion, then model error correction as a language intersection problem between a Levenshtein automaton and a context-free grammar. To decode the solutions, we then sample trees without replacement from the intersection grammar, which yields valid repairs within a certain Levenshtein distance. Finally, we rank all repairs discovered within 60 seconds by a Markov chain.

1 INTRODUCTION

Syntax repair is the problem of modifying an invalid sentence so it conforms to some grammar. Prior work has been devoted to fixing syntax errors using handcrafted heuristics. This work features a variety of approaches including rule-based systems and statistical language models. However, these techniques are often brittle, and are susceptible to misgeneralization. In a prior paper published in SPLASH, the authors sample production rules from an error correcting grammar. While theoretically sound, this technique is incomplete, i.e., not guaranteed to sample all edits within a certain Levenshtein distance, and no more. In this paper, we demonsrate it is possible to attain a significant advantage by synthesizing and scoring all repairs within a certain Levenshtein distance. Not only does this technique guarantee perfect generalization, but also helps with precision.

We take a first-principles approach making no assumptions about the sentence or grammar and focuses on correctness and end-to-end latency. Our technique is simple:

(1) We first reduce the problem of CFL recognition to Boolean tensor completion, then use that to compute the Parikh image of the CFL. This follows from a straightforward extension of the Chomsky-Schützenberger enumeration theorem.
(2) We then model syntax correction as a language intersection problem between a Levenshtein automaton and a context-free grammar, which we explicitly materialize using a specialized version of the Bar-Hillel construction to Levenshtein intersections. This greatly reduces the number of generated productions.
(3) To decode the members from the intersection grammar, we sample trees without replacement by constructing a bijection between syntax trees and the integers, then sampling integers uniformly without replacement from a finite range. This yields concrete repairs within a certain Levenshtein distance.
(4) Finally, we rank all repairs found within 60 seconds by a Markov chain.

Though simple, this technique outperforms SoTA syntax repair techniques. Its efficacy owes to the fact it does not sample edits or nor productions, but unique, fully formed repairs within a certain Levenshtein distance. It is sound and complete up to a Levenshtein bound - i.e., it will find all repairs within an arbitrary Levenshtein distance, and no more. Often the langauge of small repairs is surpisingly small compared with the langauge of all possible edits, enabling us to efficiently synthesize and score every possible solution. This offers a significant advantage over memoryless sampling techniques.

SPLASH'24, October 22-27, 2024, Pasadena, California, United States
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 EXAMPLE

Consider the following Python snippet, which contains a small syntax error:

```
def prepend(i, k, L=[]) n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

We can fix it by inserting a colon after the function definition, yielding:

```
def prepend(i, k, L=[]): n and [prepend(i - 1, k, [b] + L) for b in range(k)]
```

A careful observer will note that there is only one way to repair this Python snippet by making a single edit. In fact, many programming languages share this curious property: syntax errors with a small repair have few uniquely small repairs. Valid sentences corrupted by a few small errors rarely have many small corrections. We call such sentences *metastable*, since they are relatively stable to small perturbations, as likely to be incurred by a careless typist or novice programmer.

Let us consider a slightly more ambiguous error: `v = df.iloc(5:, 2:)`. Assuming an alphabet of just one hundred lexical tokens, this tiny statement has millions of possible two-token edits, yet only six of those possibilities are accepted by the Python parser:

(1) `v = df.iloc(5:, 2,)` (3) `v = df.iloc(5[:, 2:])` (5) `v = df.iloc[5:, 2:]`

(2) `v = df.iloc(5), 2()` (4) `v = df.iloc(5:, 2:)` (6) `v = df.iloc(5[:, 2])`

With some typing information we could easily narrow the results, but even in the absence of semantic constraints, one can probably rule out (2, 4, 6) given that `5[` and `2(` are rare bigrams in the Python language, and knowing `df.iloc` is often followed by `[`, determine (3) is most natural. This is the key insight behind our approach: we can usually locate the intended fix by exhaustively searching small repairs. As the set of small repairs is itself often small, if only we had some procedure to distinguish valid repairs, the resulting solutions could be simply ranked by naturalness.

The trouble is that any such procedure must be highly sample-efficient. We cannot afford to sample the universe of possible d-token edits, then reject invalid ones – assuming it takes just 10ms to generate and check each sample, (1-6) could take 24+ hours to find. The hardness of brute-force search grows superpolynomially with edit distance, sentence length and alphabet size. We need a more efficient procedure for sampling all and only small valid repairs.

3 PROBLEM

We can model syntax repair as a language intersection problem between a context-free language (CFL) and a regular language.

Definition 3.1 (Bounded Levenshtein-CFL reachability). Given a CFL ℓ and an invalid string $\underset{\sim}{\sigma} : \ell^{\complement}$, the BCFLR problem is to find every valid string reachable within d edits of $\underset{\sim}{\sigma}$, i.e., letting Δ be the Levenshtein metric and $L(\underset{\sim}{\sigma}, d) \coloneqq \{\sigma \mid \Delta(\underset{\sim}{\sigma}, \sigma) \leq d\}$, we seek to find $L(\underset{\sim}{\sigma}, d) \cap \ell$.

To solve this problem, we will first pose a simpler problem that only considers intersections with a finite language, then turn our attention back to BCFLR.

Definition 3.2 (Porous completion). Let $\underline{\Sigma} \coloneqq \Sigma \cup \{\_\}$, where _ denotes a hole. We denote $\sqsubseteq : \Sigma^n \times \underline{\Sigma}^n$ as the relation $\{\langle \sigma', \sigma \rangle \mid \sigma_i \in \Sigma \implies \sigma'_i = \sigma_i\}$ and the set of all inhabitants $\{\sigma' : \Sigma^+ \mid \sigma' \sqsubseteq \sigma\}$ as $\mathrm{H}(\sigma)$. Given a *porous string*, $\sigma : \underline{\Sigma}^*$ we seek all syntactically valid inhabitants, i.e., $A(\sigma) \coloneqq \mathrm{H}(\sigma) \cap \ell$.

As $A(\sigma)$ can be a large-cardinality set, we want a procedure which prioritizes natural solutions:

Definition 3.3 (Ranked repair). Given a finite language $A = L(\underset{\sim}{\sigma}, d) \cap \ell$ and a probabilistic language model $P : \Sigma^* \to [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-k maximum likelihood repairs under the language model. That is,

$$R(A, P) \coloneqq \underset{\{\boldsymbol{\sigma} \mid \boldsymbol{\sigma} \subseteq A, |\boldsymbol{\sigma}| \leq k\}}{\operatorname{argmax}} \sum_{\sigma \in \boldsymbol{\sigma}} \mathrm{P}(\sigma \mid \underset{\sim}{\sigma}, \theta) \tag{1}$$

The problem of ranked repair is typically solved by learning a distribution over string edits, however this approach can be sample-inefficient and generalize poorly to new langauges. Modeling the distribution over Σ^* forces the model to learn both syntax and stylistics. As we demonstrate, this problem can be decomposed into a bilevel objective: first retrieval, then ranking. By ensuring retrieval is sufficiently precise and exhaustive, maximizing likliehood over the set of proximal repairs can be achieved with a much weaker, syntax-oblivious language model.

Even with an extremely efficient approximate sampler for $\sigma \sim \ell_\cap$, due to the size of A, it would be intractable to sample either $\ell \cap \Sigma^n$ or $L(\underset{\sim}{\sigma}, d)$, then reject invalid ($\sigma \notin \ell$) or unreachable ($\sigma \notin L(\underset{\sim}{\sigma}, d)$) edits, and completely out of the question to sample $\sigma \sim \Sigma^*$ as do many large language models.

Instead, we will explicitly construct a grammar generating $\ell \cap L(\underset{\sim}{\sigma}, d)$, sample from it, then rerank all repairs after a fixed timeout. So long as $|\ell_\cap|$ is sufficiently small and recognizes all and only small repairs, our sampler is sure to retrieve the most natural repair and terminate quickly. Then, the problem becomes one of ranking only sampled repairs which can be completed quickly using a Markov chain.

4 METHOD

The syntax of most programming languages is context-free. Our proposed method is simple. We construct a context-free grammar representing the intersection between the langauge syntax and an automaton recognizing the Levenshtein ball of a given radius. Since CFLs are closed under intersection with regular languages, this is admissible. Three outcomes are possible:

(1) $\mathcal{G}_\cap$ is empty, in which case there is no repair within the given radius. In this case, we simply increase the radius and try again.
(2) $\mathcal{L}(\mathcal{G}_\cap)$ is small, in which case we enumerate all possible repairs. Enumeration is tractable for $\sim$ 80% of the dataset in $\leq$ 90s.
(3) $\mathcal{L}(\mathcal{G}_\cap)$ is too large to enumerate, so we sample from the intersection grammar $\mathcal{G}_\cap$. Sampling is necessary for $\sim$ 20% of the dataset.

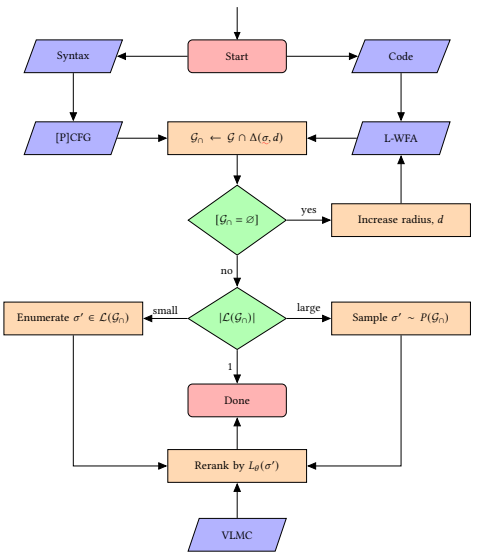


Fig. 1. Flowchart of our proposed method.

When ambiguous, we use an n-gram model to rank and return the top-k results by likelihood. This procedure is depicted in Fig. 1.

4.1 The Nominal Levenshtein Automaton

Levenshtein edits are recognized by a certain kind of automaton, known as the Levenshtein automaton. Since the original approach used by Schultz and Mihov contains cycles and epsilon transitions, we propose a modified variant which is epsilon-free, acyclic and monotone. Furthermore, we use a nominal automaton, allowing for infinite alphabets. This considerably simplifies the langauge intersection. We give an example of a small Levenshtein automaton recognizing $\Delta(\sigma : \Sigma^5, 3)$ in Fig. 2. Unlabeled arcs accept any terminal.

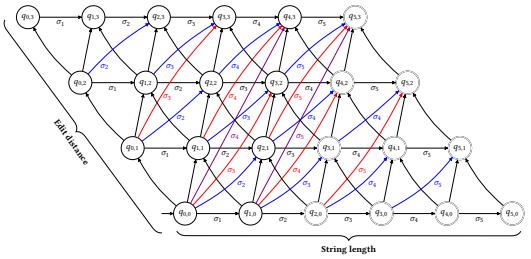


Fig. 2. NFA recognizing Levenshtein $\Delta(\sigma : \Sigma^5, 3)$.

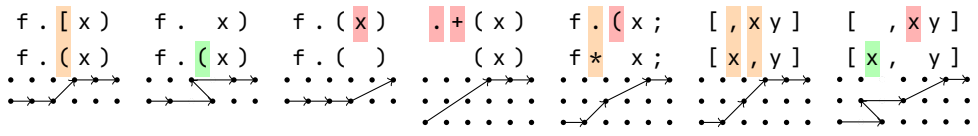
Alternatively, this transition system can be viewed as a kind of proof system. This is equivalent to the Levenshtein automaton used by Schultz and Mihov, but is more amenable to our purposes as it does not any contain ε-arcs, and uses skip connections to represent consecutive deletions.

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

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

$$\frac{}{q_{0,0} \in I}\ \textsc{Init} \qquad \frac{q_{i,j} \qquad |n-i+j| \le k}{q_{i,j} \in F}\ \textsc{Done}$$

Each arc plays a specific role. ↖ handles insertions, ↗ handles substitutions, ↖ handles insertions and ⇗ handles [consecutive] deletions of various lengths. Let us consider some illustrative cases.



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

To avoid creating multiple arcs for each insertion or deletion, we alter the following rules:

$$\frac{S(s : \Sigma) \mapsto [s \neq \sigma_i] \quad i \in [0,n] \quad j \in [1,k]}{(q_{i,j-1} \xrightarrow{S} q_{i,j}) \in \delta} \qquad \frac{S(s : \Sigma) \mapsto [s \neq \sigma_i] \quad i \in [1,n] \quad j \in [1,k]}{(q_{i-1,j-1} \xrightarrow{S} q_{i,j}) \in \delta}$$

By nominalizing the NFA, we can eliminate the creation of $e = |\Sigma| \cdot |\sigma| \cdot 2d_{\max}$ unnecessary arcs across the entire Levenshtein automaton. Thus, it is absolutely essential to first nominalize the automaton before proceeding.

4.2 Levenshtein-Bar-Hillel Construction

We now describe the Bar-Hillel construction, which generates a grammar recognizing the intersection between a regular and a context-free language, then specialize it to Levenshtein intersections.

Lemma 4.1. *For any context-free language ℓ and finite state automaton α, there exists a context-free grammar $G_\cap$ such that $\mathcal{L}(G_\cap) = \ell \cap \mathcal{L}(\alpha)$. See Bar-Hillel [7].*

Although Bar-Hillel [7] lacks an explicit construction, Beigel and Gasarch [10] construct $G_\cap$ like so:

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

This, now standard, Bar-Hillel construction applies to any CFL and REG language intersection, but generates a grammar whose cardinality is approximately $|P_\cap| = |I| \cdot |F| + |P| \cdot |\Sigma| \cdot |\sigma| \cdot 2d_{\max} + |P| \cdot |Q|^3$. Applying the BH construction directly to programming language syntax and Levenshtein automata can generate hundreds of trillions of productions for moderately sized grammars and Levenshtein automata. In this section, we describe a kind of reachability analysis that elides many unreachable productions in the case of Levenshtein intersection. We will now describe this reduction in detail.

Consider $\bowtie$, the most expensive rule. What $\bowtie$ tells us is each nonterminal in the intersection grammar, $P_\cap$, matches a substring simultaneously recognized by (1) a pair of states in the original NFA and (2) a nonterminal in the original CFG. A key observation is that $\bowtie$ considers the intersection between every possible triple, but this is a huge overapproximation for most NFAs and CFGs, as the vast majority of all state pairs and nonterminals will recognize no strings in common.

To identify these triples, we define an interval domain that soundly overapproximates the Parikh image, encoding the minimum and maximum number of terminals each nonterminal can represent. Since some intervals may be right-unbounded, we write $\mathbb{N}^* = \mathbb{N} \cup \{\infty\}$ to capture the upper bound.

Definition 4.2 (Parikh mapping of a nonterminal). Let $p : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ be the Parikh operator [29], which counts the frequency of terminals in a string. Let $\pi : V \to \{[a,b] \in \mathbb{N} \times \mathbb{N}^* \mid a \le b\}^{|\Sigma|}$ be a function returning the smallest interval such that $\forall \sigma : \Sigma^*, \forall v : V, v \Rightarrow^* \sigma \vdash p(s) \in \pi(v)$.

In other words, the Parikh mapping computes the greatest lower and least upper bound of the Parikh image over all strings in the language of a nonterminal. The infimum of a nonterminal's Parikh interval tells us how many of each terminal a nonterminal *must* generate, and the supremum tells us how many it *can* generate. Likewise, we define a similar relation over NFA state pairs:

Definition 4.3 (Parikh mapping of NFA states). We define $\pi : Q \times Q \to \{[a,b] \in \mathbb{N} \times \mathbb{N}^* \mid a \le b\}^{|\Sigma|}$ as returning the smallest interval such that $\forall \sigma : \Sigma^*, \forall q, q' : Q, q \overset{\sigma}{\Longrightarrow} q' \vdash p(\sigma) \in \pi(q,q')$.

Next, we will define a measure on Parikh intervals, which will represent the minimum total edits required to transform a string in one Parikh interval to a string in another.

Definition 4.4 (Parikh divergence). Given two Parikh intervals $\pi, \pi' : \{[a,b] \in \mathbb{N} \times \mathbb{N}^* \mid a \le b\}^{|\Sigma|}$, we define the divergence between them as $\pi \parallel \pi' = \sum_{n=1}^{|\Sigma|} \min_{(i,i') \in \pi[n] \times \pi'[n]} |i - i'|$.

Now, we know that if the Parikh divergence between two intervals exceeds the Levenshtein margin between two states in a Lev-NFA, those intervals must be incompatible as no two strings, one from each Parikh interval, can be transformed into the other in fewer than $\pi \parallel \pi'$ edits.

Definition 4.5 (Levenshtein-Parikh compatibility). Let $q = q_{h,i}, q' = q_{j,k}$ be two states in a Lev-NFA and V be a CFG nonterminal. We say that $(q, v, q') : Q \times V \times Q$ are compatible iff the Parikh divergence is bounded by the Levenshtein margin $k - i$, i.e., $v \lhd qq' \iff (\pi(v) \parallel \pi(q,q')) \le k - i$.

Finally, we define the modified Bar-Hillel construction for Levenshtein intersections as follows:

$$\frac{(A \to a) \in P \quad S(a) \quad (q \xrightarrow{S} r) \in \delta}{(qAr \to a) \in P_\cap} \qquad \frac{w \lhd pr \quad x \lhd pq \quad z \lhd qr \quad (w \to xz) \in P \quad p,q,r \in Q}{(pwr \to (pxq)(qzr)) \in P_\cap} \bowtie$$

4.3 Parsing as idempotent matrix completion

Recall that a CFG is a quadruple consisting of terminals (Σ), nonterminals (V), productions ($P\colon V \to (V \mid \Sigma)^*$), and a start symbol, ($S$). Every CFG is reducible to *Chomsky Normal Form*, $P'\colon V \to (V^2 \mid \Sigma)$, in which every P takes one of two forms, either $w \to xz$, or $w \to t$, where $w, x, z : V$ and $t : \Sigma$. For example:

$$G \coloneqq \left\{ S \to S\,S \mid (\,S\,) \mid (\,) \right\} \Longrightarrow \left\{ S \to Q\,R \mid S\,S \mid L\,R, \quad R \to), \quad L \to (, \quad Q \to L\,S \right\}$$

Given a CFG, $G' : \mathbb{G} = \langle \Sigma, V, P, S \rangle$ in CNF, we can construct a recognizer $R : \mathbb{G} \to \Sigma^n \to \mathbb{B}$ for strings $\sigma : \Sigma^n$ as follows. Let 2^V be our domain, 0 be $\varnothing$, $\oplus$ be $\cup$, and $\otimes$ be defined as:

$$X \otimes Z \coloneqq \left\{ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \right\} \tag{2}$$

If we define $\hat{\sigma}_r \coloneqq \{w \mid (w \to \sigma_r) \in P\}$, then construct a matrix with nonterminals on the superdiagonal representing each token, $M_0[r+1 = c](G', \sigma) \coloneqq \hat{\sigma}_r$ and solve for the fixpoint $M_{i+1} = M_i + M_i^2$,

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

we obtain the recognizer, $R(G', \sigma) \coloneqq [S \in \Lambda^*_\sigma] \Leftrightarrow [\sigma \in \mathcal{L}(G)]$ [1].

Since $\bigoplus_{c=1}^{n} M_{r,c} \otimes M_{c,r}$ has cardinality bounded by $|V|$, it can be represented as $\mathbb{Z}_2^{|V|}$ using the characteristic function, $\mathbb{1}$. A concrete example is shown in § **??**.

Let us consider an example with two holes, $\sigma = 1\ \_\ \_$, and the grammar being $G \coloneqq \{S \to NON, O \to + \mid \times, N \to 0 \mid 1\}$. This can be rewritten into CNF as $G' \coloneqq \{S \to NL, N \to 0 \mid 1, O \to \mid +, L \to ON\}$. Using the algebra where $\oplus = \cup$, $X \otimes Z = \left\{ w \mid \langle x, z \rangle \in X \times Z, (w \to xz) \in P \right\}$, the fixpoint $M' = M + M^2$ can be computed as follows, shown in the leftmost column:

	2^V	$\mathbb{Z}_2^{\lvert V\rvert}$	$\mathbb{Z}_2^{\lvert V\rvert} \to \mathbb{Z}_2^{\lvert V\rvert}$
M_0	$\begin{pmatrix} \{N\} & & \\ & \{N,O\} & \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square & & \\ & \square\blacksquare\blacksquare & \\ & & \square\blacksquare\blacksquare \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & & \\ & V_{1,2} & \\ & & V_{2,3} \end{pmatrix}$
M_1	$\begin{pmatrix} \{N\} & \varnothing & \\ & \{N,O\} & \{L\} \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square & \square\square\square & \\ & \square\blacksquare\blacksquare & \blacksquare\square\square \\ & & \square\blacksquare\blacksquare \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & \\ & V_{1,2} & V_{1,3} \\ & & V_{2,3} \end{pmatrix}$
M_∞	$\begin{pmatrix} \{N\} & \varnothing & \{S\} \\ & \{N,O\} & \{L\} \\ & & \{N,O\} \end{pmatrix}$	$\begin{pmatrix} \square\blacksquare\square & \square\square\square & \square\square\square\blacksquare \\ & \square\blacksquare\blacksquare & \blacksquare\square\square\square \\ & & \square\blacksquare\blacksquare\square \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & V_{0,3} \\ & V_{1,2} & V_{1,3} \\ & & V_{2,3} \end{pmatrix}$

The same procedure can be translated, without loss of generality, into the bit domain ($\mathbb{Z}_2^{|V|}$) using a lexicographic ordering, however these both are recognizers. That is to say, $[S \in V_{0,3}] \Leftrightarrow [V_{0,3,3} =$

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

■] ⇔ $[A(\sigma) \neq \varnothing]$. Since $V_{0,3} = \{S\}$, we know there is at least one $\sigma' \in A(\sigma)$, but M_∞ does not reveal its identity.

In order to extract the inhabitants, we can translate the bitwise procedure into an equation with free variables. Here, we can encode the idempotency constraint directly as $M = M^2$. We first define $X \boxtimes Z = [X_2 \wedge Z_1, \bot, \bot, X_1 \wedge Z_0]$ and $X \boxplus Z = [X_i \vee Z_i]_{i \in [0,|V|)}$. 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 $V_{0,2}, V_{1,3}$ as follows:

$$\begin{aligned} V_{0,2} &= V_{0,j} \cdot V_{j,2} = V_{0,1} \boxtimes V_{1,2} && (3)\\ &= [L \in V_{0,2}, \bot, \bot, S \in V_{0,2}] && (4)\\ &= [O \in V_{0,1} \wedge N \in V_{1,2}, \bot, \bot, N \in V_{0,1} \wedge L \in V_{1,2}] && (5)\\ &= [V_{0,1,2} \wedge V_{1,2,1}, \bot, \bot, V_{0,1,1} \wedge V_{1,2,0}] && (6) \end{aligned}$$

$$\begin{aligned} V_{1,3} &= V_{1,j} \cdot V_{j,3} = V_{1,2} \boxtimes V_{2,3} && (7)\\ &= [L \in V_{1,3}, \bot, \bot, S \in V_{1,3}] && (8)\\ &= [O \in V_{1,2} \wedge N \in V_{2,3}, \bot, \bot, N \in V_{1,2} \wedge L \in V_{2,3}] && (9)\\ &= [V_{1,2,2} \wedge V_{2,3,1}, \bot, \bot, V_{1,2,1} \wedge V_{2,3,0}] && (10) \end{aligned}$$

Now we solve for the corner entry $V_{0,3}$ by taking the bitwise dot product between the first row and last column, yielding:

$$\begin{aligned} V_{0,3} &= V_{0,j} \cdot V_{j,3} = V_{0,1} \boxtimes V_{1,3} \boxplus V_{0,2} \boxtimes V_{2,3} && (11)\\ &= [V_{0,1,2} \wedge V_{1,3,1} \vee V_{0,2,2} \wedge V_{2,3,1}, \bot, \bot, V_{0,1,1} \wedge V_{1,3,0} \vee V_{0,2,1} \wedge V_{2,3,0}] && (12) \end{aligned}$$

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

$$\begin{aligned} V_{0,3,3} &= V_{0,1,1} \wedge V_{1,3,0} \vee V_{0,2,1} \wedge V_{2,3,0} && (13)\\ &= V_{0,1,1} \wedge (V_{1,2,2} \wedge V_{2,3,1}) \vee V_{0,2,1} \wedge \bot && (14)\\ &= V_{0,1,1} \wedge V_{1,2,2} \wedge V_{2,3,1} && (15)\\ &= [N \in V_{0,1}] \wedge [O \in V_{1,2}] \wedge [N \in V_{2,3}] && (16) \end{aligned}$$

Now we know that $\sigma = 1\ \underline{\mathrm{O}}\ \underline{\mathrm{N}}$ is a valid solution, and therefor we can take the product $\{1\} \times \hat{\sigma}_r^{-1}(O) \times \hat{\sigma}_r^{-1}(N)$ to recover the admissible set, yielding $A(\sigma) = \{1+0, 1+1, 1\times 0, 1\times 1\}$. In this case, since G is unambiguous, there is only one parse tree satisfying $V_{0,|\sigma|,|\sigma|}$, but in general, there can be multiple valid parse trees, in which case we can decode them incrementally.

4.4 A Pairing Function for Breadth-Bounded Binary Trees

Now that we have a method for parsing with matrix completion, we will translate this to the tree domain, where the semiring algebra will be defined over indexed forests. Then we will show how to sample trees.

We will now describe a technique for sampling trees from the intersection grammar, representing distinct, fully formed repairs. When the number of choices is sufficiently constrained, this can be sampled without replacement, or otherwise with replacement using a PCFG. Once we have obtained the intersection grammar, we solve for all inhabitants using a matrix fixedpoint recurrence.

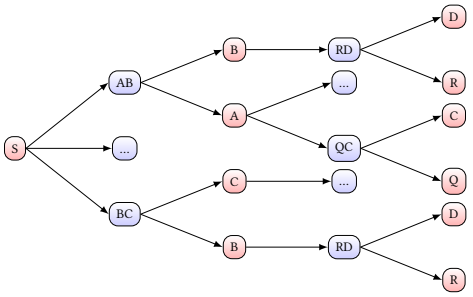


Fig. 3. A partial $\mathbb{T}_2$ for the grammar $P = \{S \to BC \mid \ldots \mid AB, B \to RD \mid \ldots, A \to QC \mid \ldots\}$.

We define an algebraic data type $\mathbb{T}_3 = (V \cup \Sigma) \rightharpoonup \mathbb{T}_2$ where $\mathbb{T}_2 = (V \cup \Sigma) \times (\mathbb{N} \rightharpoonup \mathbb{T}_2 \times \mathbb{T}_2)$[2]. Morally, we can think of $\mathbb{T}_2$ as an implicit set of possible trees sharing the same root, and $\mathbb{T}_3$ as a dictionary of possible $\mathbb{T}_2$ values indexed by possible roots, given by a specific CFG under a finite-length porous string. We construct $\hat{\sigma}_r = \Lambda(\sigma_r)$ as follows:

$$\Lambda(s : \underline{\Sigma}) \mapsto \begin{cases} \bigoplus_{s \in \Sigma} \Lambda(s) & \text{if } s \text{ is a hole,} \\ \left\{\mathbb{T}_2\left(w, \left[\langle \mathbb{T}_2(s), \mathbb{T}_2(\varepsilon)\rangle\right]\right) \mid (w \to s) \in P\right\} & \text{otherwise.} \end{cases}$$

This initializes the superdiagonal entries, enabling us to compute the fixpoint M_∞ by redefining $\oplus, \otimes : \mathbb{T}_3 \times \mathbb{T}_3 \to \mathbb{T}_3$ as:

$$X \oplus Z \mapsto \bigcup_{k \in \pi_1(X \cup Z)} \left\{k \Rightarrow \mathbb{T}_2(k, x \cup z) \mid x \in \pi_2(X \circ k), z \in \pi_2(Z \circ k)\right\}$$

$$X \otimes Z \mapsto \bigoplus_{(w \to xz) \in P} \left\{\mathbb{T}_2\left(w, \left[\langle X \circ x, Z \circ z\rangle\right]\right) \mid x \in \pi_1(X), z \in \pi_1(Z)\right\}$$

These operators group subtrees by their root nonterminal, then aggregate their children. Instead of tracking sets, each Λ now becomes a dictionary indexed by the root nonterminal, which can be sampled by obtaining $(\Lambda^*_\sigma \circ S) : \mathbb{T}_2$, then recursively choosing twins as we describe in § **??**, or without replacement via enumeration as described in § 4.4.

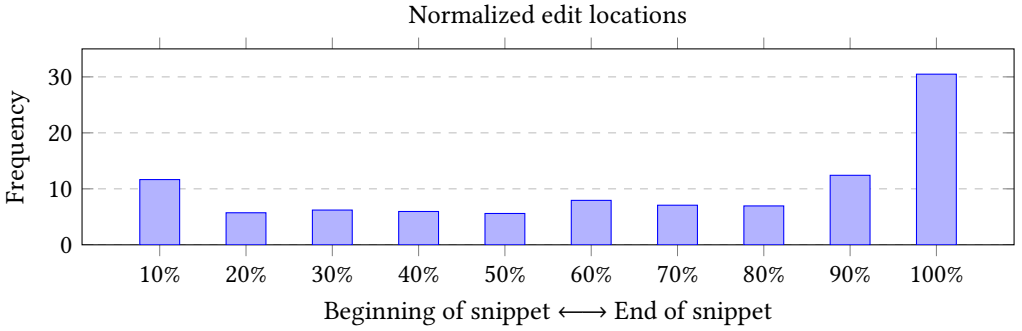
Given a probabilistic CFG whose productions indexed by each nonterminal are decorated with a probability vector $\mathbf{p}$ (this may be uniform in the non-probabilistic case), we define a tree sampler $\Gamma : (\mathbb{T}_2 \mid \mathbb{T}_2^2) \rightsquigarrow \mathbb{T}$ which recursively samples children according to a Multinoulli distribution:

$$\Gamma(T) \mapsto \begin{cases} \Gamma\left(\text{Multi}\left(\texttt{children}(T), \mathbf{p}\right)\right) & \text{if } T : \mathbb{T}_2 \\ \left\langle \Gamma\left(\pi_1(T)\right), \Gamma\left(\pi_2(T)\right)\right\rangle & \text{if } T : \mathbb{T}_2 \times \mathbb{T}_2 \end{cases}$$

This is closely related to the generating function for the ordinary Boltzmann sampler from analytic combinatorics,

$$\Gamma C(x) \mapsto \begin{cases} \text{Bern}\left(\frac{A(x)}{A(x)+B(x)}\right) \to \Gamma A(x) \mid \Gamma B(x) & \text{if } \mathcal{C} = \mathcal{A} + \mathcal{B} \\ \left\langle \Gamma A(x), \Gamma B(x)\right\rangle & \text{if } \mathcal{C} = \mathcal{A} \times \mathcal{B} \end{cases}$$

[2]Given a $T : \mathbb{T}_2$, we may also refer to $\pi_1(T), \pi_2(T)$ as $\texttt{root}(T)$ and $\texttt{children}(T)$ respectively, where children are pairs of conjoined twins.



however unlike Duchon et al. [**?**], our work does not depend on rejection to guarantee exact-size sampling, as all trees contained in $\mathbb{T}_2$ will necessarily be the same width.

The type $\mathbb{T}_2$ of all possible trees that can be generated by a CFG in Chomksy Normal Form corresponds to the fixpoints of the following recurrence, which tells us that each $\mathbb{T}_2$ can be a terminal, nonterminal, or a nonterminal and a sequence consisting of nonterminal pairs and their two children:

$$L(p) = 1 + pL(p) \qquad P(a) = \Sigma + V + VL\left(V^2 P(a)^2\right)$$

Given a $\sigma : \underline{\Sigma}$, we construct $\mathbb{T}_2$ from the bottom-up, and sample from the top-down. Depicted below is a partial $\mathbb{T}_2$, where red nodes are `roots` and blue nodes are `children`:

The number of binary trees inhabiting a single instance of $\mathbb{T}_2$ is sensititive to the number of nonterminals and rule expansions in the grammar. To obtain the total number of trees with breadth n, we abstractly parse the porous string using the algebra defined in § **??**, letting $T = \Lambda^*_{\underline{\sigma}} \circ S$, and compute the total number of trees using the recurrence:

$$|T : \mathbb{T}_2| \mapsto \begin{cases} 1 & \text{if } T \text{ is a leaf,} \\ \sum_{\langle T_1, T_2 \rangle \in \texttt{children}(T)} |T_1| \cdot |T_2| & \text{otherwise.} \end{cases}$$

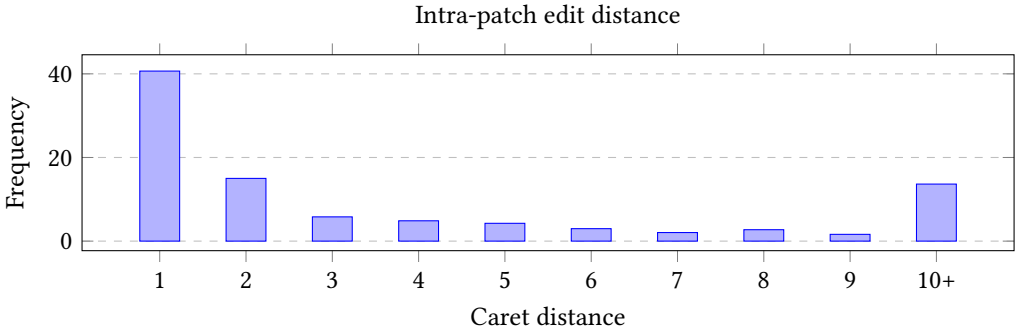
To sample all trees in a given $T : \mathbb{T}_2$ uniformly without replacement, we then construct a modular pairing function $\varphi : \mathbb{T}_2 \to \mathbb{Z}_{|T|} \to \texttt{BTree}$, which is defined as follows:

$$\varphi(T : \mathbb{T}_2, i : \mathbb{Z}_{|T|}) \mapsto \begin{cases} \left\langle \texttt{BTree}\big(\texttt{root}(T)\big), i \right\rangle & \text{if } T \text{ is a leaf,} \\ \text{Let } b = |\texttt{children}(T)|, & \\ \quad q_1, r = \left\langle \lfloor \frac{i}{b} \rfloor, i \pmod b \right\rangle, & \\ \quad lb, rb = \texttt{children}[r], & \\ \quad T_1, q_2 = \varphi(lb, q_1), & \\ \quad T_2, q_3 = \varphi(rb, q_2) \text{ in} & \\ \left\langle \texttt{BTree}\big(\texttt{root}(T), T_1, T_2\big), q_3 \right\rangle & \text{otherwise.} \end{cases}$$

Then, instead of sampling trees, we can simply sample integers uniformly without replacement from $\mathbb{Z}_{|T|}$ using the construction defined in § **??**, and lazily decode them into trees.

5 DATASET

The StackOverflow dataset is comprised of 500k Python code snippets, each of which has been annotated with a human repair. We depict the normalized edit loations relative to the snippet length below.



Likewise, we can plot the number of tokens between edits within each patch:

6 EXPERIMENTAL SETUP

To evaluate our model, we primarily use 5,600 pairs of (broken, fixed) Python code snippets from Wong et al.'s StackOverflow dataset [35] shorter than 40 lexical tokens, whose minimized patch sizes are shorter than four lexical tokens ($|\Sigma| = 50, |\sigma| \leq 40, \Delta(\sigma, \ell) < 4$), and adapt the Python grammar from SeqParse. Minimization uses the Delta debugging [37] technique to isolate the smallest lexical patch that repairs a broken Python snippet.

In the first set of experiments, we uniformly sample without replacement from the Levenshtein edit ball using a LFSR over $\mathbb{Z}_2^m$, and measure the precision@k of our repair procedure against human repairs of varying edit distances and latency cutoffs. This provides a baseline for the relative density of the admissible set, and an upper bound on the latency of attaining a given precision.

In the second set of experiments, we use an adaptive sampler that stochastically resamples edits using a Dirichlet process. Repairs are scored and placed into a ranked buffer, from which new edits are resampled with frequency relative to their perplexity, and additive noise is introduced. The adaptive sampler is described in further detail in §**??**.

To train the scoring function, we use a length-5 variable-order Markov (VOM) chain implemented using a count-min sketch based on Apache Datasketches [5]. Training on 55 million StackOverflow tokens from the BIFI [36] dataset took roughly 10 minutes, after which calculating perplexity is nearly instantaneous. Sequences are scored using negative log likelihood with Laplace smoothing and our evaluation measures the precision@{1, 5, 10, All} for samples at varying latency cutoffs.

Both sets of experiments were conducted on 40 Intel Skylake cores running at 2.4 GHz, with 16 GB of RAM, running bytecode compiled for JVM 17.0.2. We measure the precision using abstract lexical matching, following the Seq2Parse [31] evaluation, and give a baseline for their approach on the same dataset.

7 EVALUATION

We consider the following research questions for our evaluation:

- **RQ 1**: What are the characteristics of the human repair? (Levenshtein edit distance, distance between edits etc)
- **RQ 2**: How do we compare with SoTA approaches? (e.g., Seq2Parse, BIFI, OrdinalFix, et al.)
- **RQ 3**: Ablation study over design choices? (search vs. sampling, timing cutoffs, ngram models, etc.)

For our evaluation, we use the StackOverflow dataset from [22]. We preprocess the dataset to lexicalize both the broken and fixed code snippets, then filter the dataset by length and edit distance,

in which all Python snippets whose broken form is fewer than 80 lexical tokens and whose human fix is under four Levenshtein edits is retained.

For our first experiment, we run the sampler until the human repair is detected, then measure the number of samples required to draw the exact human repair across varying Levenshtein radii.

Fig. 4. Sample efficiency of LBH sampler at varying Levenshtein radii.

Next, measure the precision at various ranking cutoffs for varying wall-clock timeouts. Here, P@{k=1, 5, 10, All} indicates the percentage of syntax errors with a human repair of $\Delta = \{1, 2, 3, 4\}$ edits found in $\leq p$ seconds that were matched within the top-k results, using an n-gram likelihood model.

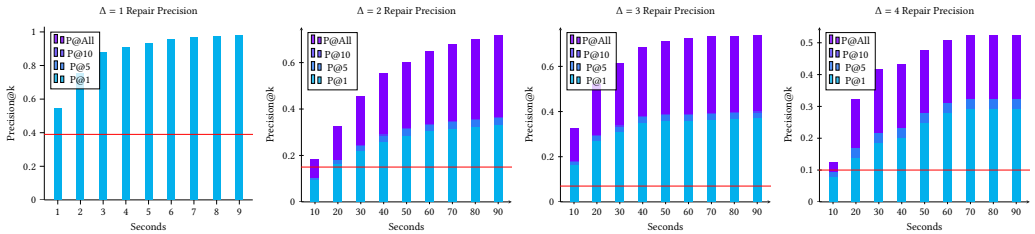


Fig. 5. Human repair benchmark. Note the y-axis across different edit distance plots has varying ranges.

7.1 Old Evaluation

We evaluate Tidyparse along three primary axes: latency, throughput, and accuracy on a dataset of human repairs. Our intention here is to show that Tidyparse is competitive with a large language model (roughly, a deep circuit) that is slow but highly sample-efficient with a small language model (roughly, a shallow circuit) that is fast but less sample-efficient.

Large language models typically take between several hundred milliseconds and several seconds to infer a repair. The output is not guaranteed to be syntactically valid, and may require more than one sample to discover a valid repair. In contrast, Tidyparse can discover thousands of repairs in the same duration, all of which are guaranteed to be syntactically valid. Furthermore, if a valid repair exists within a certain number of edits, it will eventually be found.

To substantiate these claims, we conduct experiments measuring:

- the average worst-case time to discover a human repair across varying sizes, i.e., average latency to discover a repair with edit distance d.
- the average accuracy at varying latency cutoffs, i.e., average precision@k at latency cutoff t.
- the average repair throughput across varying CPU cores, i.e., average number of admissible repairs discovered per second over the repair length.
- the relative throughput versus a uniform sampler, i.e., average number of admissible repairs discovered per second divided by the uniform sampler's throughput

7.2 Uniform sampling benchmark

Below, we plot the precision of the uniform sampling procedure described in §?? against human repairs of varying edit distances and latency cutoffs. Repairs discovered before timeout expiration are reranked by tokenwise perplexity then compared using an exact lexical match with the human repair at or below rank k. We note that the uniform sampling procedure is not intended to be used

in practice, but rather provides a baseline for the empirical density of the admissible set, and an upper bound on the latency required to attain a given precision.

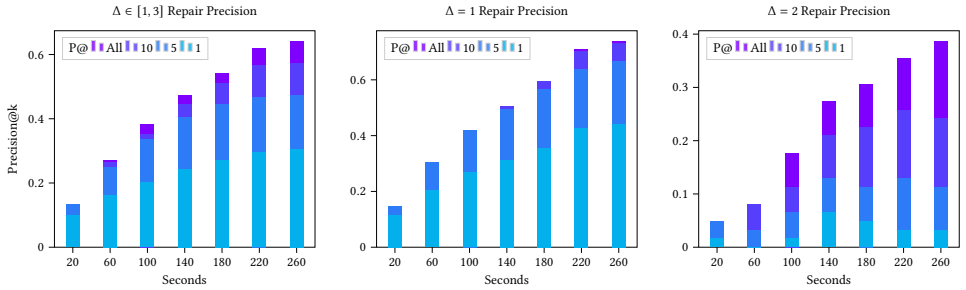


Fig. 6. Human repair benchmark. Note the y-axis across different edit distance plots has varying ranges.

Despite the high-latency, this demonstrates a uniform prior with post-timeout reranking is still able to achieve competitive precision@k using a relatively cheap ranking metric. This suggests that we can use the metric to bias the sampler towards more likely repairs, which we will now do.

7.3 Repair with an adaptive sampler

In the following benchmark, we measure the precision@k of our repair procedure against human repairs of varying edit distances and latency cutoffs, using an adaptive resampling procedure described in §??. This sampler maintains a buffer of successful repairs ranked by perplexity and uses stochastic local search to resample edits within a neighborhood. Initially, edits are sampled uniformly at random. Over time and as the admissible set grows, it prioritizes edits nearby low-perplexity repairs. This technique offers a significant advantage in the low-latency setting.

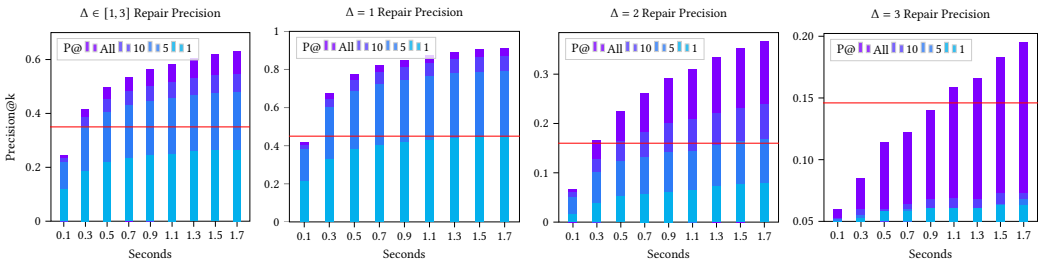


Fig. 7. Adaptive sampling repairs. The red line indicates Seq2Parse precision@1 on the same dataset. Since it only supports generating one repair, we do not report precision@k or the intermediate latency cutoffs.

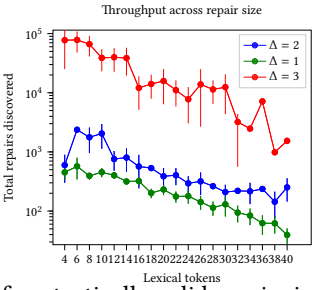
We also evaluate Seq2Parse on the same dataset. Seq2Parse only supports precision@1 repairs, and so we only report Seq2parse precision@1 from the StackOverflow benchmark for comparison. Unlike our approach which only produces syntactically correct repairs, Seq2Parse also produces syntactically incorrect repairs and so we report the percentage of repairs matching the human repair for both our method and Seq2Parse. Seq2Parse latency varies depending on the length of the repair, averaging 1.5s for $\Delta = 1$ to 2.7s for $\Delta = 3$, across the entire StackOverflow dataset.

While adapting sampling is able to saturate the admissible set for 1- and 2-edit repairs before the timeout elapses, 3-edit throughput is heavily constrained by compute around 16 lexical tokens, when

Python's Levenshtein ball has a volume of roughly 6×10^8 edits. This bottleneck can be relaxed with a longer timeout or additional CPU cores. Despite the high computational cost of sampling multi-edit repairs, our precision@all remains competitive with the Seq2Parse neurosymbolic baseline at the same latency. We provide some qualitative examples of repairs in Table ??.

7.4 Throughput benchmark

End-to-end throughput varies significantly with the edit distance of the repair. Some errors are trivial to fix, while others require a large number of edits to be sampled before a syntactically valid edit is discovered. We evaluate throughput by sampling edits across invalid strings $|\sigma| \leq 40$ from the StackOverflow dataset of varying length, and measure the total number of syntactically valid edits discovered, as a function of string length and language edit distance $\Delta \in [1, 3]$. Each trial is terminated after 10 seconds, and the experiment is repeated across 7.3k total repairs. Note the y-axis is log-scaled, as the number of admissible repairs increases sharply with language edit distance. Our approach discovers a large number of syntactically valid repairs in a relatively short amount of time, and is able to quickly saturate the admissible set for 1- and 2-edit repairs before timeout. As the Seq2Parse baseline is unable to generate more than one syntactically valid repair per string, we do not report its throughput.



7.5 Timing benchmark

7.6 Synthetic repair benchmark

In addition to the StackOverflow dataset, we also evaluate our approach on two datasets containing synthetic strings generated by a Dyck language, and bracketing errors of synthetic and organic provenance in organic source code. The first dataset contains length-50 strings sampled from various Dyck languages, i.e., the Dyck language containing n different types of balanced parentheses. The second contains abstracted Java and Python source code mined from GitHub repositories. The Dyck languages used in the remaining experiments are defined by the following context-free grammar(s):

```
Dyck-1 -> ( ) | ( Dyck-1 ) | Dyck-1 Dyck-1
    Dyck-2 -> Dyck-1 | [ ] | ( Dyck-2 ) | [ Dyck-2 ] | Dyck-2 Dyck-2
    Dyck-3 -> Dyck-2 | { } | ( Dyck-3 ) | [ Dyck-3 ] | { Dyck-3 } | Dyck-3 Dyck
        -3
```

In experiment (1a), we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number of indices in $[0, |\sigma|)$ with holes and measure the average time required to decode ten syntactically-admissible repairs across 100 trial runs. In experiment (1b), we sample a random valid string as before, but delete p tokens at random and rather than provide their location(s), ask our model to solve for both the location(s) and repair by sampling uniformly from all n-token HCs, then measure the total time required to decode the first admissible repair. Note the logarithmic scale on the y-axis.

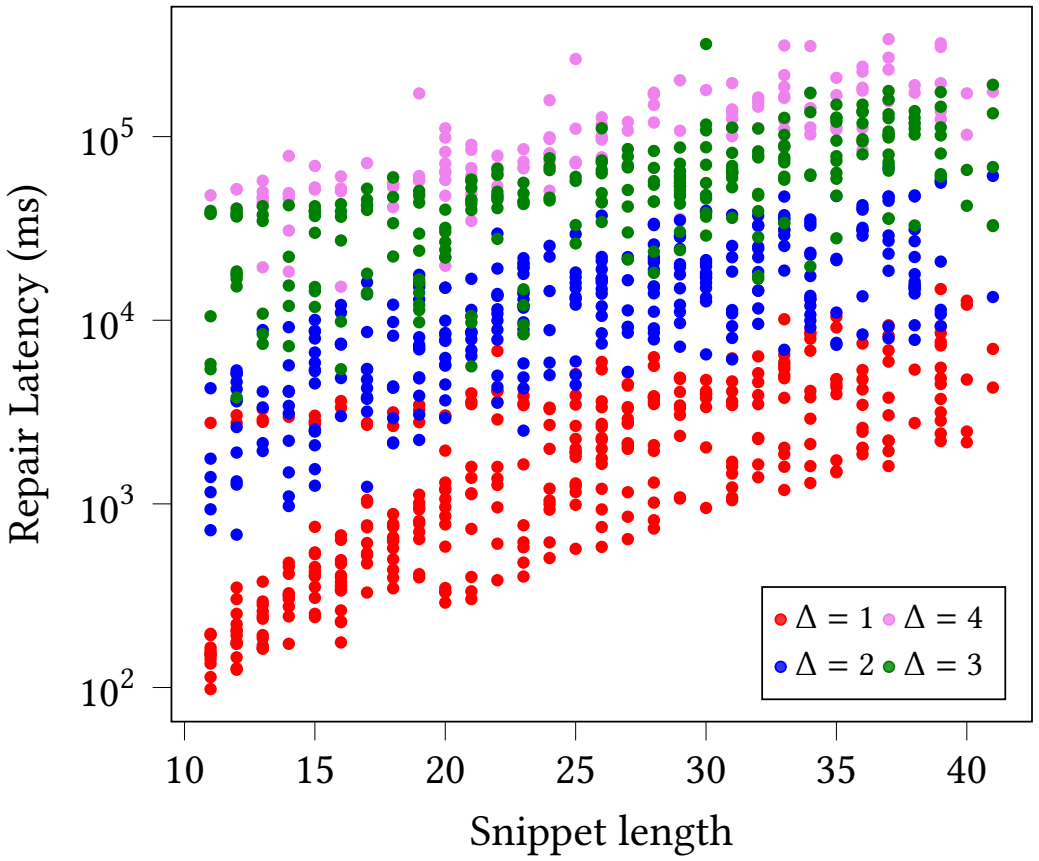


Fig. 8. End-to-end repair timings across snippet length.

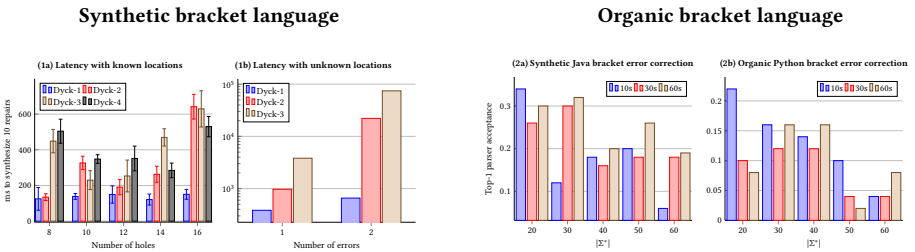


Fig. 9. Benchmarking bracket correction latency and accuracy across two bracketing languages, one generated from Dyck-n, and the second uses an abstracted source code snippet with imbalanced parentheses.

In the second set of experiments, we analyze bracketing errors in a dataset of Java and Python code snippets mined from open-source repositories on GitHub using the Dyck-nw[3], in which all source code tokens except brackets are replaced with a `w` token. For Java (2a), we sample valid single-line statements with bracket nesting more than two levels deep, synthetically delete one

[3]Using the Dyck-n grammar augmented with a single additional production, `Dyck-1` → `w` | `Dyck-1`. Contiguous non-bracket characters are substituted with a single placeholder token, `w`, and restored verbatim after bracket repair.

bracket uniformly at random, and repair using Tidyparse, then take the top-1 repair after t seconds, and validate using ANTLR's Java 8 parser. For Python (2b), we sample invalid code fragments uniformly from the imbalanced bracket category of the Break-It-Fix-It (BIFI) dataset [36], a dataset of organic Python errors, which we repair using Tidyparse, take the top-1 repair after t seconds, and validate repairs using Python's `ast.parse()` method. Since the Java and Python datasets do not have a ground-truth human fix, we report the percentage of repairs that are accepted by the language's official parser for repairs generated under a fixed time cutoff. Although the Java and Python datasets are not directly comparable, we observe that Tidyparse can detect and repair a significant fraction of bracket errors in both languages with a relatively unsophisticated grammar.

8 RELATED WORK

Three important questions arise when repairing syntax errors: (1) is the program broken in the first place? (2) if so, where are the errors located? (3) how should those locations then be altered? In the case of syntax correction, those questions are addressed by three related research areas, (1) parsing, (2) language equations and (3) repair. We survey each of those areas in turn.

8.1 Parsing

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

General CFL parsing allows ambiguity (non-unique trees) and can be formulated as a dynamic programming problem, as shown by Cocke-Younger-Kasami (CYK) [30], Earley [20] and others. These parsers have roughly cubic complexity with respect to the length of the input string.

As shown by Valiant [34], Lee [25] 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 unlocks the door to a wide range of complexity-theoretic and practical speedups to CFL recognition and fast general parsing algorithms.

Okhotin (2001) [28] extends CFGs with language conjunction in *conjunctive grammars*, followed by Zhang & Su (2017) [38] who apply conjunctive language reachability to dataflow analysis.

8.2 Language equations

Language equations are a powerful tool for reasoning about formal languages and their inhabitants. First proposed by Ginsburg et al. [21] for the ALGOL language, language equations are essentially systems of inequalities with variables representing *holes*, i.e., unknown values, in the language or grammar. Solutions to these equations can be obtained using various fixpoint techniques, yielding members of the language. This insight reveals the true algebraic nature of CFLs and their cousins.

Being an algebraic formalism, language equations naturally give rise to a kind of calculus, vaguely reminiscent of Leibniz' and Newton's. First studied by Brzozowski [12, 13] and Antimirov [4], one can take the derivative of a language equation, yielding another equation. This 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, 27] and automata minimization [11]. In our setting, differentiation corresponds to code completion.

In this paper, we restrict our attention to language equations over context-free and weakly context-sensitive languages, whose variables coincide with edit locations in the source code of a computer program, and solutions correspond to syntax repairs. Although prior work has studied the use of language equations for parsing [27], to our knowledge they have never previously been considered for the purpose of code completion or syntax error correction.

8.3 Syntax repair

In finite languages, syntax repair corresponds to spelling correction, a more restrictive and largely solved problem. Schulz and Stoyan [32] construct a finite automaton that returns the nearest dictionary entry by Levenshtein edit distance. Though considerably simpler than syntax correction, their work shares similar challenges and offers insights for handling more general repair scenarios.

When a sentence is grammatically invalid, parsing grows more challenging. Like spelling, the problem is to find the minimum number of edits required to transform an arbitrary string into a syntactically valid one, where validity is defined as containment in a (typically) context-free language. Early work, including Irons [23] and Aho [2] propose a dynamic programming algorithm to compute the minimum number of edits required to fix an invalid string. Prior work on error correcting parsing only considers the shortest edit(s), and does not study multiple edits over the Levenshtein ball. Furthermore, the problem of actually generating the repairs is not well-posed, as there are usually many valid strings that can be obtained within a given number of edits. We instead focus on bounded Levenshtein reachability, which is the problem of finding useful repairs within a fixed Levenshtein distance of the broken string, which requires language intersection.

8.4 Classical program synthesis

There is related work on string constraint solving in the constraint programing literature, featuring solvers like CFGAnalyzer and HAMPI [24], which consider bounded context free grammars and intersections thereof. Axelson et al. (2008) [6] has some work on incremental SAT encoding but does not exploit the linear-algebraic structure of parsing, conjunctive reachability nor provide real-time guarantees. D'Antoni et al. (2014) introduces *symbolic automata* [17], a generalization of finite automata which allow infinite alphabets and symbolic expressions over them. In none of the constraint programming literature we surveyed do any of the approaches employ matrix-based parsing, and therefor do not enjoy the optimality guarantees of Valiant's parser. Our solver can handle context-free and conjunctive grammars with finite alphabets and does not require any special grammar encoding. The matrix encoding makes it particularly amenable to parallelization.

8.5 Error correcting codes

Our work focuses on errors arising from human factors in computer programming, in particular *syntax error correction*, which is the problem of fixing partially corrupted programs. Modern research on error correction, however, can be traced back to the early days of coding theory when researchers designed *error-correcting codes* (ECCs) to denoise transmission errors induced by external interference, e.g., collision with a high-energy proton, manipulation by an adversary or even typographical mistake. In this context, *code* can be any logical representation for communicating information between two parties (such as a human and a computer), and an ECC is a carefully-designed scheme which ensures that even if some portion of the message should become corrupted, one can still recover the original message by solving a linear system of equations. When designing ECCs, one typically assumes a noise model over a certain event space, such as the Hamming [18, 33] or Levenshtein [8, 9] balls, from which we draw inspiration for our work.

8.6 Neural program repair

The recent success of deep learning has lead to a variety of work on neural program repair [3, 15, 19]. These approaches typically employ Transformer-based neural language models (NLMs) and model the problem as a sequence-to-sequence transformation. Although recent work on circuit lower bounds have cast doubt on the ability of Transformers to truly learn formal languages [14, 26], expressivity aside, these models have been widely adopted for practical program repair tasks. In

particular, two papers stand out being most closely related to our own: Break-It-Fix-It (BIFI) [36] and Seq2Parse [31]. BIFI adapts techniques from semi-supervised machine translation to generate synthetic errors in clean code and fixes them. This reduces the amount of pairwise training data, but may generalize poorly to natural errors. Seq2Parse combines a transformer-based model with an augmented version of the Early parser to suggest error rules, but only suggests a single repair. Our work differs from both in that we suggest multiple repairs with much lower latency, do not require a pairwise repair dataset, and can fix syntax errors in any language with a well-defined grammar. We note our approach is complementary to existing work in neural program repair, and may be used to generate synthetic repairs for training or employ a NLM model to rank its repairs.

8.7 Uniform sampling benchmark

Below, we plot the precision of the uniform sampling procedure described in §?? against human repairs of varying edit distances and latency cutoffs. Repairs discovered before the latency cutoff are reranked based on their tokenwise perplexity and compared for an exact lexical match with the human repair at or below rank k. We note that the uniform sampling procedure is not intended to be used in practice, but rather provides a baseline for the empirical density of the admissible set, and an upper bound on the latency required to attain a given precision.

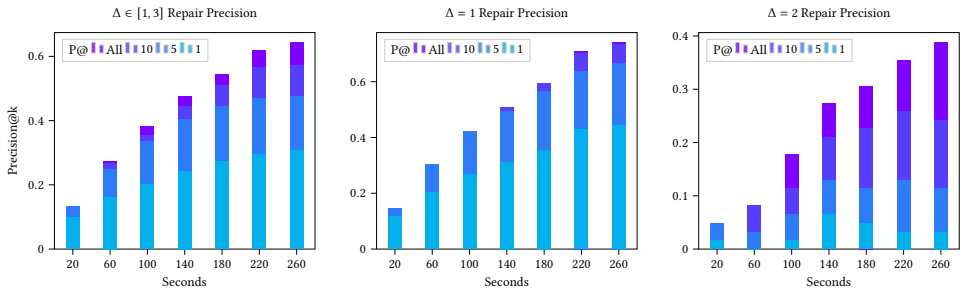


Fig. 10. Human repair benchmark. Note the y-axis across different edit distance plots has varying ranges.

Despite the high-latency, this demonstrates a uniform prior with post-timeout reranking is still able to achieve competitive precision@k using a relatively cheap ranking metric. This suggests that we can use the metric to bias the sampler towards more likely repairs, which we will now do.

8.8 Repair with an adaptive sampler

In the following benchmark, we measure the precision@k of our repair procedure against human repairs of varying edit distances and latency cutoffs, using an adaptive resampling procedure described in §??. This sampler maintains a buffer of successful repairs ranked by perplexity and uses stochastic local search to resample edits within a neighborhood. Initially, edits are sampled uniformly at random. Over time and as the admissible set grows, it prioritizes edits nearby low-perplexity repairs. This technique offers a significant advantage in the low-latency setting.

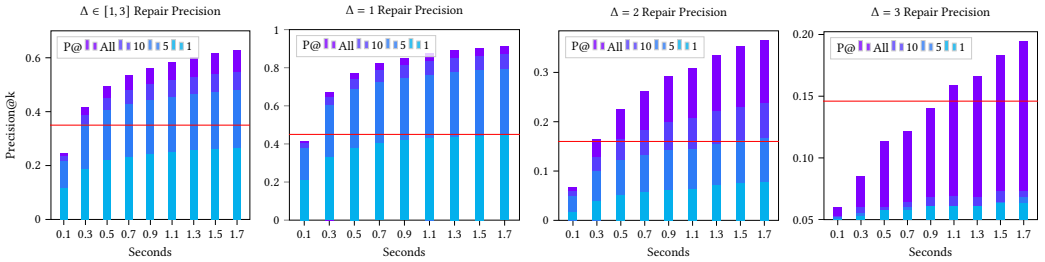
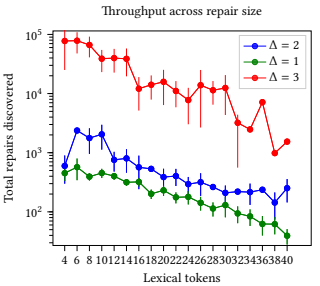


Fig. 11. Adaptive sampling repairs. The red line indicates Seq2Parse precision@1 on the same dataset. Since it only supports generating one repair, we do not report precision@k or the intermediate latency cutoffs.

We also evaluate Seq2Parse on the same dataset. Seq2Parse only supports precision@1 repairs, and so we only report Seq2parse precision@1 from the StackOverflow benchmark for comparison. Unlike our approach which only produces syntactically correct repairs, Seq2Parse also produces syntactically incorrect repairs and so we report the percentage of repairs matching the human repair for both our method and Seq2Parse. Seq2Parse latency varies depending on the length of the repair, averaging 1.5s for $\Delta = 1$ to 2.7s for $\Delta = 3$, across the entire StackOverflow dataset.

While adapting sampling is able to saturate the admissible set for 1- and 2-edit repairs before the timeout elapses, 3-edit throughput is heavily constrained by compute around 16 lexical tokens, when Python's Levenshtein ball has a volume of roughly 6×10^8 edits. This bottleneck can be relaxed with a longer timeout or additional CPU cores. Despite the high computational cost of sampling multi-edit repairs, our precision@all remains competitive with the Seq2Parse neurosymbolic baseline at the same latency. We provide some qualitative examples of repairs in Table **??**.

8.9 Throughput benchmark



End-to-end throughput varies significantly with the edit distance of the repair. Some errors are trivial to fix, while others require a large number of edits to be sampled before a syntactically valid edit is discovered. We evaluate throughput by sampling edits across invalid strings $|\sigma| \leq 40$ from the StackOverflow dataset of varying length, and measure the total number of syntactically valid edits discovered, as a function of string length and language edit distance $\Delta \in [1, 3]$. Each trial is terminated after 10 seconds, and the experiment is repeated across 7.3k total repairs. Note the y-axis is log-scaled, as the number of admissible repairs increases sharply with language edit distance. Our approach discovers a large number of syntactically valid repairs in a relatively short amount of time, and is able to quickly saturate the admissible set for 1- and 2-edit repairs before timeout. As the Seq2Parse baseline is unable to generate more than one syntactically valid repair per string, we do not report its throughput.

8.10 Synthetic repair benchmark

In addition to the StackOverflow dataset, we also evaluate our approach on two datasets containing synthetic strings generated by a Dyck language, and bracketing errors of synthetic and organic provenance in organic source code. The first dataset contains length-50 strings sampled from various Dyck languages, i.e., the Dyck language containing n different types of balanced parentheses. The second contains abstracted Java and Python source code mined from GitHub repositories. The Dyck languages used in the remaining experiments are defined by the following context-free grammar(s):

```
Dyck-1 -> ( ) | ( Dyck-1 ) | Dyck-1 Dyck-1
    Dyck-2 -> Dyck-1 | [ ] | ( Dyck-2 ) | [ Dyck-2 ] | Dyck-2 Dyck-2
    Dyck-3 -> Dyck-2 | { } | ( Dyck-3 ) | [ Dyck-3 ] | { Dyck-3 } | Dyck-3 Dyck
        -3
```

In experiment (1a), we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number of indices in $[0, |\sigma|)$ with holes and measure the average time required to decode ten syntactically-admissible repairs across 100 trial runs. In experiment (1b), we sample a random valid string as before, but delete p tokens at random and rather than provide their location(s), ask our model to solve for both the location(s) and repair by sampling uniformly from all n-token HCs, then measure the total time required to decode the first admissible repair. Note the logarithmic scale on the y-axis.

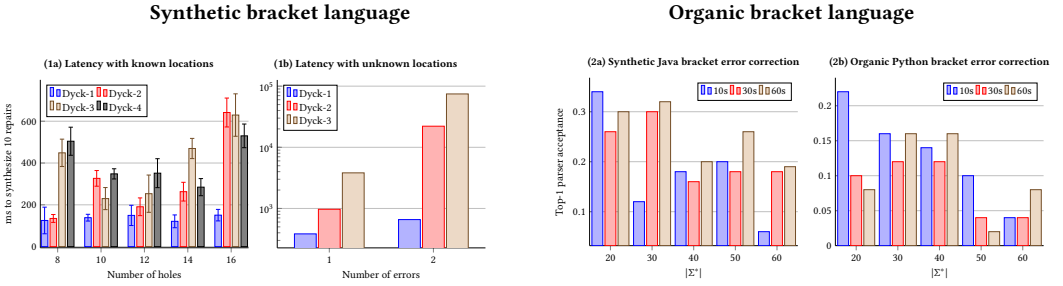


Fig. 12. Benchmarking bracket correction latency and accuracy across two bracketing languages, one generated from Dyck-n, and the second uses an abstracted source code snippet with imbalanced parentheses.

In the second set of experiments, we analyze bracketing errors in a dataset of Java and Python code snippets mined from open-source repositories on GitHub using the Dyck-nw[4], in which all source code tokens except brackets are replaced with a `w` token. For Java (2a), we sample valid single-line statements with bracket nesting more than two levels deep, synthetically delete one bracket uniformly at random, and repair using Tidyparse, then take the top-1 repair after t seconds, and validate using ANTLR's Java 8 parser. For Python (2b), we sample invalid code fragments uniformly from the imbalanced bracket category of the Break-It-Fix-It (BIFI) dataset [36], a dataset of organic Python errors, which we repair using Tidyparse, take the top-1 repair after t seconds, and validate repairs using Python's `ast.parse()` method. Since the Java and Python datasets do not have a ground-truth human fix, we report the percentage of repairs that are accepted by the language's official parser for repairs generated under a fixed time cutoff. Although the Java and Python datasets are not directly comparable, we observe that Tidyparse can detect and repair a significant fraction of bracket errors in both languages with a relatively unsophisticated grammar.

9 DISCUSSION

The main lesson we draw from our experiments is that it is possible to leverage compute to compete with large language models on practical program repair tasks. Though extremely sample-efficient, their size comes at the cost of higher latency, and domain adaptation requires fine-tuning or retraining on pairwise repairs. Our approach uses a tiny grammar and a relatively cheap ranking

[4]Using the Dyck-n grammar augmented with a single additional production, `Dyck-1` → `w` | `Dyck-1`. Contiguous non-bracket characters are substituted with a single placeholder token, `w`, and restored verbatim after bracket repair.

metric to achieve comparable accuracy at the same latency. This allows us to repair errors in languages with little to no training data and provides far more flexibility and controllability.

Our primary insight leading to SoTA precision@{5,10} is that repairs are typically concentrated near a small number of edit locations, and by biasing the search toward previously successful edit locations, we can achieve a significant speedup over a naïve search. We note this heuristic may not be applicable to all grammars, and it may be possible to construct less natural counterexamples where this heuristic fails, although we consider these unlikely in practice.

Latency can vary depending on many factors including string length and grammar size, and critically the Levenshtein edit distance. This can be an advantage because in the absence of any contextual or statistical information, syntax and Levenshtein edits are often sufficiently constrained to determine a small number of valid repairs. It is also a limitation because as the number of edits grows, the admissible set grows rapidly and the number of valid repairs may become too large to be useful without a good metric, depending on the language and source code snippet under repair.

Although possible to further reduce constraint size by preprocessing techniques such as those posed in §**??**, the length of the string under repair is by far the dominating factor in formula size. For this reason, the isolation problem posed in Def. **??** is a critical obstacle to overcome. A general solution would allow us to collapse well-formed substrings and handle much larger code fragments than are currently supported. We consider this a promising direction for future work.

Tidyparse in its current form has several other technical shortcomings: firstly, it does not incorporate any neural language modeling technology at present, an omission we hope to address. Training a language model to predict likely repair locations and rank admissible results could lead to lower overall latency and more natural repairs. We also hope to explore the use of Metropolis-Hastings and determinantal point processes to encourage sampling diversity.

Secondly, our current method does not specialize language intersection to the grammar family, nor employ Bar-Hillel's [7] construction for REG-CFL intersection, which would lead to a more efficient encoding of Levenshtein-CFL reachability. Furthermore, considering recent extensions of Boolean matrix-based parsing to linear context-free rewriting systems (LCFRS) [16], it may be feasible to search through richer language families within the SAT solver without employing an external stochastic search to generate and validate candidate repairs.

Lastly and perhaps most significantly, Tidyparse does not incorporate any semantic constraints, so its repairs whilst syntactically admissible, are not guaranteed to be semantically valid. This can be partly alleviated by filtering the results through an incremental compiler or linter, however, the latency introduced may be non-negligible. It is also possible to encode type-based semantic constraints into the solver and we intend to explore this direction more fully in future work.

We envision a few primary use cases for our tool: (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. Featuring a grammar editor and built-in SAT solver, Tidyparse helps developers navigate the language design space, visualize syntax trees, debug parsing errors and quickly generate simple examples and counterexamples for benchmarking and testing.

10 CONCLUSION

The great compromise in program synthesis is one of efficiency versus expressiveness. The more expressive a language, the more concise and varied the programs it can represent, but the harder those programs are to synthesize without resorting to domain-specific heuristics. Likewise, the simpler a language is to synthesize, the weaker its concision and expressive power.

Most existing work on program synthesis has focused on general λ-calculi, or narrow languages such as finite sets or regular expressions. The former are too expressive to be efficiently synthesized or verified, whilst the latter are too restrictive to be useful. In our work, we focus on context-free and mildly context-sensitive grammars, which are expressive enough to capture a variety of useful programming language features, but not so expressive as to be unsynthesizable.

The second great compromise in program synthesis is that of reusability versus specialization. In programming, as in human communications, there is a vast constellation of languages, each requiring specialized generators and interpreters. Are these languages truly irreconcilable? Or, as Noam Chomsky argues, are these merely dialects of a universal language? *Synthesis* then, might be a misnomer, and more aptly called *recognition*, in the analytic tradition.

In our work, we argue these two compromises are not mutually exclusive, but complementary and reciprocal. Programs and the languages they inhabit are indeed synthetic, but can be analyzed and reused in the metalanguage of context-free grammars closed under conjunction. Not only does this admit an efficient synthesis algorithm, but allows users to introduce additional constraints without breaking compositionality, one of the most sacred tenets in programming language design.

Over the last few years, there has been a surge of progress in applying language models to write programs. That work is primarily based on methods from differential calculus and continuous optimization, leading to the so-called *naturalness hypothesis*, which suggests programming languages are not so different from natural ones. In contrast, programming language theory takes the view that languages are essentially discrete and finitely-generated sets, although perhaps uncomputably large ones, governed by logical calculi. Programming, thus viewed, is more like a mathematical exercise in constraint satisfaction, an oft-overlooked but unavoidable aspect of translating abstract ideas into computing machinery. These constraints naturally arise at various stages of syntax validation, type-checking and runtime verification, and help to ensure programs fulfill their intended purpose.

As our work shows, not only is linear algebra over finite fields an expressive language for probabilistic inference, but also an efficient framework for inference on languages themselves. Borrowing analysis techniques from multilinear algebra and tensor completion in the machine learning setting, we develop an equational theory that allows us to translate various decision problems on formal languages into a system of inequalities over finite fields. As a practical consequence, this means we can efficiently encode a number of problems in parsing, code completion and program repair using SAT solvers, which are known to be highly efficient, scalable and flexible to domain-specific constraints. We demonstrate the effectiveness of our approach in a variety of practical scenarios, including code completion and program repair for linear conjunctive languages, and show that our approach is competitive with state-of-the-art methods in terms of both accuracy and efficiency. In contrast with LL and LR-style parsers, our technique can recover partial forests from invalid strings, and handle arbitrary conjunctive languages. In future work, we hope to extend our method to more natural grammars like PCFG, LCFRS and other mildly context-sensitive languages.

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

Despite its computational complexity, the design of the tool itself is relatively simple. Tidyparse accepts a linear conjunctive language and a string. If the string is valid, it returns the parse forest, otherwise, it returns a set of repairs, ordered by their perplexity. Tidyparse compiles the grammar and candidate string onto a discrete dynamical system using an extended version of Valiant's algorithm and solves for its fixedpoints using an incremental SAT solver. By allowing the string to

contain holes, repairs may contain either concrete tokens or nonterminals, which can be expanded by the user or a neural-guided search procedure to produce a valid program. This approach to parsing has many advantages, enabling us to repair syntax errors, correct typos and recover from errors in real time, as well as being provably sound and complete with respect to the grammatical specification. It is also compatible with neural program synthesis and repair techniques and shares the same masked language modeling (MLM) target used by Transformer-based LLMs.

We have implemented our approach as an IDE plugin and demonstrated its viability as a practical tool for realtime programming. A considerable amount of effort was devoted to supporting fast error correction functionality. Tidyparse is capable of generating repairs for invalid code in a range of practical languages with very little downstream language integration required. We plan to continue expanding its grammar and autocorrection functionality to cover a broader range of languages and hope to conduct a more thorough user study to validate its effectiveness.

REFERENCES

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

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

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

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

[5] Apache Software Foundation. 2022. Apache DataSketches. https://datasketches.apache.org/.

[6] Roland Axelsson, Keijo Heljanko, and Martin Lange. 2008. Analyzing context-free grammars using an incremental SAT solver. In Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35. Springer, 410–422.

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

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

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

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

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

[13] 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.

[14] David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter Bounds on the Expressivity of Transformer Encoders. arXiv preprint arXiv:2301.10743 (2023).

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

[16] Shay B Cohen and Daniel Gildea. 2016. Parsing linear context-free rewriting systems with fast matrix multiplication. Computational Linguistics 42, 3 (2016), 421–455.

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

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

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

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

[21] Seymour Ginsburg and H Gordon Rice. 1962. Two families of languages related to ALGOL. Journal of the ACM (JACM) 9, 3 (1962), 350–371.
[22] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE). IEEE, 837–847. https://doi.org/10.1145/2902362
[23] E. T. Irons. 1963. An Error-Correcting Parse Algorithm. Commun. ACM 6, 11 (nov 1963), 669–673. https://doi.org/10.1145/368310.368385
[24] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In Proceedings of the eighteenth international symposium on Software testing and analysis. 105–116.
[25] 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
[26] 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.
[27] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. ACM sigplan notices 46, 9 (2011), 189–195.
[28] Alexander Okhotin. 2001. Conjunctive grammars. Journal of Automata, Languages and Combinatorics 6, 4 (2001), 519–535.
[29] Rohit J. Parikh. 1966. On Context-Free Languages. J. ACM 13, 4 (oct 1966), 570–581. https://doi.org/10.1145/321356.321364
[30] Itiroo Sakai. 1961. Syntax in universal translation. In Proceedings of the International Conference on Machine Translation and Applied Language Analysis.
[31] 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.
[32] Klaus U Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition 5 (2002), 67–85.
[33] Michalis K Titsias and Christopher Yau. 2017. The Hamming ball sampler. J. Amer. Statist. Assoc. 112, 520 (2017), 1598–1611.
[34] 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
[35] 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.
[36] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In International Conference on Machine Learning. PMLR, 11941–11952.
[37] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. ACM SIGSOFT Software Engineering Notes 27, 6 (2002), 1–10.
[38] 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.