Syntax Repair as Idempotent Tensor Completion

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 taking a grammar and a malformed string, and modifying the string so it conforms to the grammar. Prior work has been devoted to fixing syntax errors using handcrafted heuristics. We take a first-principles approach that makes no assumptions about the string or grammar and focuses on accuracy 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 is surprisingly effective and competitive with SoTA syntax repair techniques. Its efficiency owes to the fact that it does not sample edits, 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 fixed Levenshtein distance, and no more.

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,)`

(2) `v = df.iloc(5), 2()`

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

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

(5) `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 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 simply enumerate all possible repairs. Enumeration is tractable for $\sim 80\%$ of the Python dataset in ≤ 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 Python dataset.

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

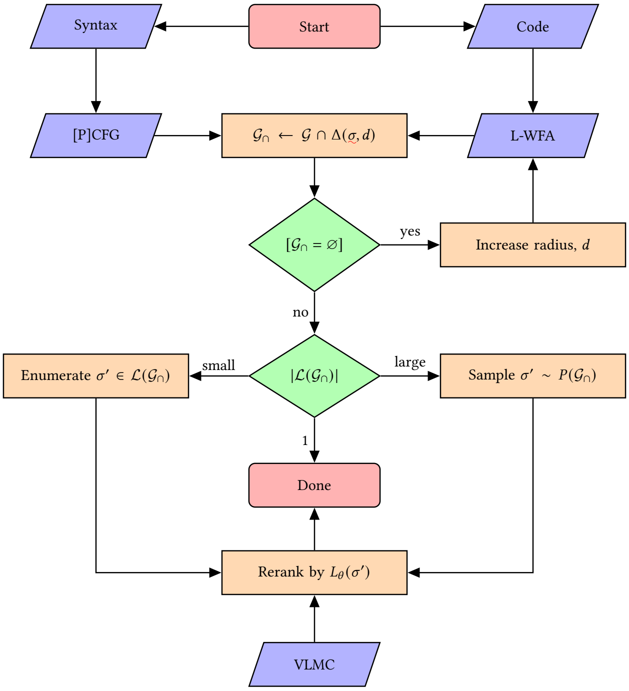

Syntax
Start
Code
[P]CFG
$\mathcal{G}_\cap \leftarrow \mathcal{G} \cap \Delta(\sigma, d)$
L-WFA
$[\mathcal{G}_\cap = \varnothing]$
yes
Increase radius, d
no
Enumerate $\sigma' \in \mathcal{L}(\mathcal{G}_\cap)$
small
$|\mathcal{L}(\mathcal{G}_\cap)|$
large
Sample $\sigma' \sim P(\mathcal{G}_\cap)$
1
Done
Rerank by $L_\theta(\sigma')$
VLMC

Fig. 1. Flowchart of our proposed method.

3.1 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.

3.2 The Bar-Hillel Construction

The Bar-Hillel construction is a general method for obtaining the context-free grammar representing the intersection of a context-free language and a regular language. We will now present the epsilon-free version of the Bar-Hillel construction used in our work.[1]

3.3 The Levenshtein-Bar-Hillel-Parikh Reduction

The standard BH construction applies to any CFL and REG language. While straightforward, the general method can generate hundreds of trillions of productions for moderately sized grammars and Levenshtein automata. Our method considerably simplifies this process by eliminating the need to materialize most of those productions, and is the key to making our approach tractable.

To achieve this, we precompute upper and lower Parikh bounds for every terminal and integer range of the string, which we call the Parikh map. This construction soundly overapproximates the minimum and maximum number of terminals that can be derived from a given nonterminal in a

[1] Clemente Pasti has a version of the BH construction that supports epsilon transitions, but is slightly more complicated.

bounded-length string, and is used to prune the search space. We will now describe this reduction in detail.

4 PROBLEM

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

Definition 4.1 (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 4.2 (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 admissible inhabitants, i.e., $A(\sigma) \coloneqq \mathrm{H}(\sigma) \cap \ell$.

$A(\sigma)$ is often a large-cardinality set, so we want a procedure which returns the most likely members first, without exhaustive enumeration. More precisely,

Definition 4.3 (Ranked repair). Given a finite language $\ell_\cap = L(\underset{\sim}{\sigma}, d) \cap \ell$ and a probabilistic language model $P_\theta : \Sigma^* \to [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-k repairs by likelihood under the language model. That is,

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

We want a procedure $\hat{R}$, minimizing $\mathbb{E}_{G,\sigma}\left[D_{\mathrm{KL}}(\hat{R} \parallel R)\right]$ and wallclock runtime.

Even with an extremely efficient approximate sampler for $\sigma \sim \ell_\cap$, due to the large cardinality of $\mathcal{L}(G) \cap \Sigma^n$ and $L(\underset{\sim}{\sigma}, d)$, it would be intractable to sample either set, 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 for the language $\mathcal{L}(G) \cap L(\underset{\sim}{\sigma}, d)$, sample from it without replacement, then rerank all consistient repairs after a fixed timeout. As long as $|\ell_\cap|$ is sufficiently small and recognizes the true repair, our sampler is sure to retrieve it and terminate quickly. Then, the problem becomes one of ranking all sampled repairs which can be completed quickly using a Markov chain.

4.1 Background

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:

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

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:

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

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

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

we obtain the recognizer, $R(G',\sigma) := [S \in \Lambda_\sigma^*] \Leftrightarrow [\sigma \in \mathcal{L}(G)]$ [2].

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 § 4.2.

4.2 Example

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

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

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, $[S \in V_{0,3}] \Leftrightarrow [V_{0,3,3} = \blacksquare] \Leftrightarrow [A(\sigma) \neq \varnothing]$. 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:

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

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

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

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

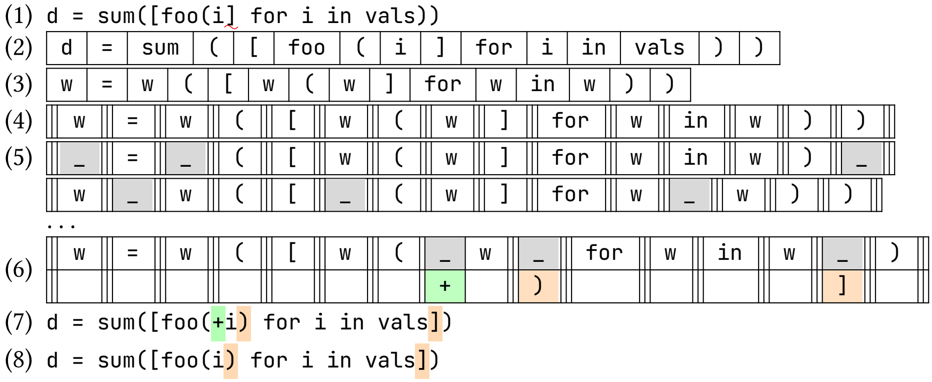
$$
\begin{aligned}
V_{0,3} &= V_{0,j} \cdot V_{j,3} = V_{0,1} \boxtimes V_{1,3} \boxplus V_{0,2} \boxtimes V_{2,3} && (11)\\
&= [V_{0,1,2} \land V_{1,3,1} \lor V_{0,2,2} \land V_{2,3,1}, \bot, \bot, V_{0,1,1} \land V_{1,3,0} \lor V_{0,2,1} \land V_{2,3,0}] && (12)
\end{aligned}
$$

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:

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

Now we know that $\sigma = 1\ \underline{O}\ \underline{N}$ 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.

The question naturally arises, where should one put the holes? One solution is to interleave ε between every token as $\sigma_\varepsilon \coloneqq (\varepsilon^c \sigma_i)_{i=1}^n\, \varepsilon^c$, augment the grammar to admit ε^+, then sample holes without replacement from all possible locations. Below we illustrate this procedure on a single Python snippet.



```
(1) d = sum([foo(i] for i in vals))
(2) | d | = | sum | ( | [ | foo | ( | i | ] | for | i | in | vals | ) | ) |
(3) | w | = | w | ( | [ | w | ( | w | ] | for | w | in | w | ) | ) |
(4) || w || = || w || ( || [ || w || ( || w || ] || for || w || in || w || ) || ) ||
(5) || _ || = || _ || ( || [ || w || ( || w || ] || for || w || in || w || ) || _ ||
    || w || _ || w || ( || [ || _ || ( || w || ] || for || w || _ || w || ) || ) ||
...
(6) || w || = || w || ( || [ || w || ( || _ | w || _ || for || w || in || w || _ || ) ||
    ||   ||   ||   ||   ||   ||   ||   || + |   || ) ||     ||   ||    ||   || ] ||   ||
(7) d = sum([foo(+i) for i in vals])
(8) d = sum([foo(i) for i in vals])
```

The initial broken string, `d = sum([foo(i] for i in vals))` (1), is first tokenized using a lexer to obtain the sequence in (2).

4.3 Bar-Hillel Construction

Manually generating the edits is a more controllable way to synthesize edits, but can be unnecessarily expensive if the goal is to synthesize all edits within a fixed edit distance. The second approach is more efficient, but requires generating a large grammar. We now describe the Bar-Hillel construction, which generates a grammar recognizing the intersection between a finite automaton, and then use the grammar to generate the edits without enumerating holes.

Definition 4.4. A finite state automaton is a tuple $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, and $I, F \subseteq Q$ are the set of initial and final states, respectively.

LEMMA 4.5. *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 [1].*

Beigel and Gasarch [?] provide one explicit way to construct $G_\cap$:

$$\frac{q \in I \quad r \in F}{(S \to qSr) \in P_\cap} \qquad \frac{(q \overset{a}{\to} r) \in \delta}{(qar \to a) \in P_\cap} \qquad \frac{(w \to xz) \in P \qquad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap}\ \text{ẅ}$$

Conveniently, the Levenshtein ball is recognized by a nondeterministic finite automaton (NFA). From Lemma 4.5, we know the intersection of any context-free language and NFA is context-free, and therefor we can construct a single context-free grammar $G_\cap$ which recognizes and generates the admissible set.

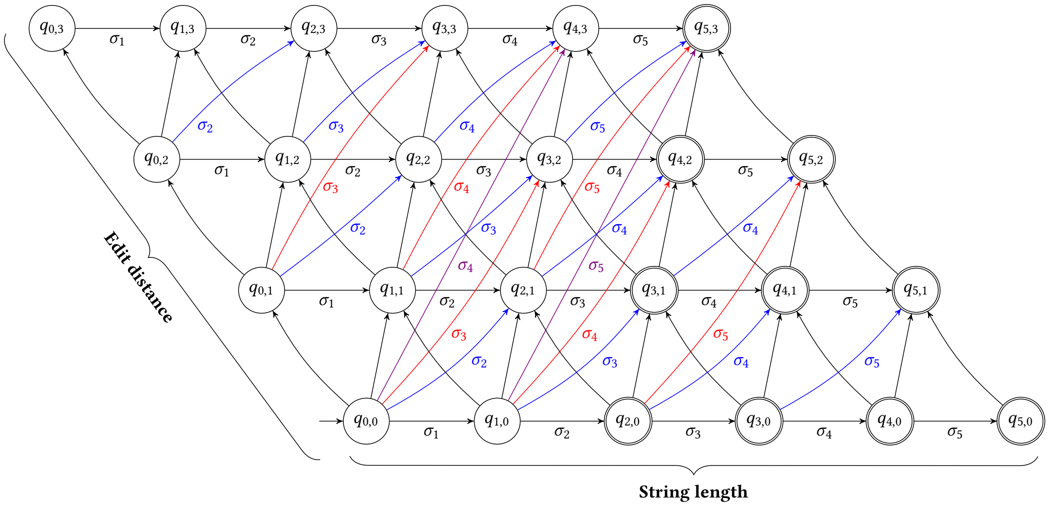


Fig. 2. Automaton recognizing Levenshtein $\Delta(\sigma : \Sigma^5, 3)$ reachability. Unlabeled arcs accept any terminal.

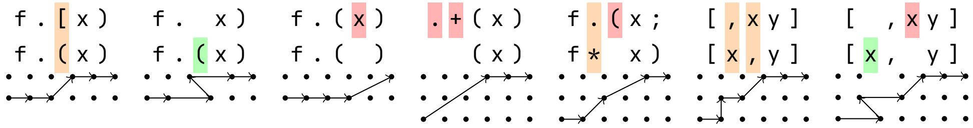
Alternatively, this transition system can be viewed as a kind of proof system.

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

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

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

Each arc plays a specific role. ↖ handles insertions, ↗ handles substitutions, ↖ handles insertions and ↗ handles deletions of various lengths. Let us consider some illustrative examples.



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

4.4 Levenshtein Bar-Hillel Specialization

The standard Bar-Hillel construction applies to all context-free grammars and automata, but can be specialized to intersections with Levenshtein automata to avoid creating many unnecessary productions. In this section, we describe a kind of reachability analysis that removes all nonterminals unreachable from the start symbol or terminals in a certain distance.

Let us define a relation over the set of nonterminals in a grammar which computes the upper and lower bounds of the Parikh image. This will tell us the minimum and maximum number of symbols each nonterminal can represent.

Definition 4.6 (Parikh interval). Let $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$ be the Parikh vector [6], which counts the number of times each terminal appears in a string. We define a function $\pi : V \times \Sigma \rightarrow \mathbb{N}_{[a,b]}$, where $a, b : \mathbb{N} \cup \{\infty\}$, that computes the greatest lower and least upper bound of the Parikh image over all strings in the language of a nonterminal, i.e., $\forall s : \Sigma^{[l,h]}, v : V$ such that $v \Rightarrow^* s$, we have $p(s,) \in \pi(v, s)$.

Roughly, the infimum of a nonterminal's Parikh interval tells us how many of each terminal it must generate, and the supremum tells us how many it can generate. We then intersect the Parikh intervals with all triples in the Levenshtein automaton to obtain the subset of $Q \times Q \times Q \times P$ to which the rule $\ddot{\text{w}}$ applies.

Specifically, we compute Parikh intervals generated by every path though the Levenshtein automaton, then intersect the Parikh intervals for the candidate nonterminals in question. Suppose we have a $p, q, r : Q$ and $w \rightarrow xz$, then check if $[\pi(q, q') \cap \pi(v) = \varnothing]$ for all pwr, pxq, qzr. If so, we can immediately rule out this tuple.

4.5 Semiring Algebras

There are a number of alternate semirings which can be used to solve for $A(\sigma)$. A first approach propagates the values from the bottom-up, while mapping nonterminals to lists of strings. Letting $D = V \rightarrow \mathcal{P}(\Sigma^*)$, we define $\oplus, \otimes : D \times D \rightarrow D$. Initially, we construct $M_0[r + 1 = c] = \hat{\sigma}_r = p(\sigma_r)$ as follows:

$$p(s : \Sigma) \mapsto \{w \mid (w \rightarrow s) \in P\} \text{ and } p(\_) \mapsto \bigcup_{s \in \Sigma} p(s) \tag{17}$$

Like the recognizer defined in § 4.1, $p(\cdot)$ constructs elements of the superdiagonal, then we compute the fixpoint using the algebra:

$$X \oplus Z \mapsto \left\{w \overset{+}{\Rightarrow} (X \circ w) \cup (Z \circ w) \mid w \in \pi_1(X \cup Z)\right\} \tag{18}$$

$$X \otimes Z \mapsto \bigoplus_{w,x,z} \left\{w \overset{+}{\Rightarrow} (X \circ x)(Z \circ z) \mid (w \rightarrow xz) \in P, x \in X, z \in Z\right\} \tag{19}$$

After the fixpoint M_∞ is attained, the solutions can be read off via $\Lambda_\sigma^* \circ S$. The issue here is an exponential growth in cardinality when eagerly computing the transitive closure, which grows impractical for even small strings and grammars.

This encoding can be made more compact by propagating an algebraic data type $\mathbb{T}_3 = (V \cup \Sigma) \rightharpoonup \mathbb{T}_2$ where $\mathbb{T}_2 = (V \cup \Sigma) \times (\mathbb{N} \rightharpoonup \mathbb{T}_2 \times \mathbb{T}_2)^3$. 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 $\hat{\sigma}_r = \dot{p}(\sigma_r)$ as follows:

[3] Hereinafter, given a concrete $T : \mathbb{T}_2$, we shall refer to $\pi_1(T), \pi_2(T)$ as `root`(T) and `children`(T) respectively.

$$\dot{p}(s : \underline{\Sigma}) \mapsto \begin{cases} \bigoplus_{s\in\Sigma} \dot{p}(s) & \text{if } s \text{ is a hole,} \\ \Big\{\mathbb{T}_2\big(w, \big[\langle \mathbb{T}_2(s), \mathbb{T}_2(\varepsilon)\rangle\big]\big) \mid (w \to s) \in P\Big\} & \text{otherwise.}\end{cases} \tag{20}$$

We then compute the fixpoint M_∞ by redefining $\oplus, \otimes : \mathbb{T}_3 \times \mathbb{T}_3 \to \mathbb{T}_3$ as follows:

$$X \oplus Z \mapsto \bigcup_{k\in\pi_1(X\cup Z)} \Big\{k \Rightarrow \mathbb{T}_2(k, x \cup z) \mid x \in \pi_2(X \circ k), z \in \pi_2(Z \circ k)\Big\} \tag{21}$$

$$X \otimes Z \mapsto \bigoplus_{(w\to xz)\in P} \Big\{\mathbb{T}_2\big(w, \big[\langle X \circ x, Z \circ z\rangle\big]\big) \mid x \in \pi_1(X), z \in \pi_1(Z)\Big\} \tag{22}$$

Decoding trees from $(\Lambda^*_\sigma \circ S) : \mathbb{T}_2$ becomes a straightforward matter of enumeration using a recursive choice function that emits a sequence of binary trees generated by the CFG. We define this construction more precisely in § 4.6.

4.6 A Pairing Function for Breadth-Bounded Binary Trees

The type $\mathbb{T}_2$ of all possible trees that can be generated by a CFG in Chomksy Normal Form is identified by a recurrence relation:

$$L(p) = 1 + pL(p) \qquad\qquad P(a) = V + aL\big(V^2 P(a)^2\big) \tag{23}$$

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 can take the intersection between a CFG and the regular language, $\mathcal{L}(G_\cap) \coloneqq \mathcal{L}(\mathcal{G}) \cap \Sigma^n$, abstractly parse the string containing all holes, let $T = \Lambda^*_{\underline{\sigma}} \circ S$, and compute the total number of trees using the following recurrence:

$$|T : \mathbb{T}_2| \mapsto \begin{cases} 1 & \text{if } T \text{ is a leaf,} \\ \sum_{\langle T_1, T_2\rangle \in \texttt{children}(T)} |T_1| \cdot |T_2| & \text{otherwise.}\end{cases} \tag{24}$$

To sample all trees in a given $T : \mathbb{T}_2$ uniformly without replacement, we first define a pairing function $\varphi^{-1} : \mathbb{T}_2 \to \mathbb{Z}_{|T|} \to \texttt{BTree}$ as follows:

$$\varphi^{-1}(T : \mathbb{T}_2, i : \mathbb{Z}_{|T|}) \mapsto \begin{cases} \Big\langle \texttt{BTree}\big(\texttt{root}(T)\big), i\Big\rangle & \text{if } T \text{ is a leaf,} \\ \text{Let } b = |\texttt{children}(T)|, & \\ \quad q_1, r = \big\langle \lfloor \frac{i}{b} \rfloor, i \pmod{b}\big\rangle, & \\ \quad lb, rb = \texttt{children}[r], & \\ \quad T_1, q_2 = \varphi^{-1}(lb, q_1), & \\ \quad T_2, q_3 = \varphi^{-1}(rb, q_2) \text{ in} & \\ \Big\langle \texttt{BTree}\big(\texttt{root}(T), T_1, T_2\big), q_3\Big\rangle & \text{otherwise.}\end{cases} \tag{25}$$

Then, instead of sampling trees, we can simply sample integers uniformly without replacement from $\mathbb{Z}_{|T|}$ using the construction defined in 4.8, and lazily decode them into trees.

4.7 Complexity

Let us consider some loose bounds on the complexity of BCFLR. To do, we first consider the complexity of computing language-edit distance, which is a lower-bound on BCFLR complexity.

Definition 4.7. Language edit distance (LED) is the problem of computing the minimum number of edits required to transform an invalid string into a valid one, where validity is defined as containment in a context-free language, $\ell : \mathcal{L}$, i.e., $\Delta^*(\underset{\sim}{\sigma}, \ell) := \min_{\sigma \in \ell} \Delta(\underset{\sim}{\sigma}, \sigma)$, and Δ is the Levenshtein distance. LED is known to have subcubic complexity [2].

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

Lemma 4.8. *For any nonempty language $\ell : \mathcal{L}$ and invalid string $\underset{\sim}{\sigma} : \Sigma^n \cap \bar{\ell}$, there exists an $(\tilde{\sigma}, m)$ such that $\tilde{\sigma} \in \ell \cap \Sigma^m$ and $0 < \Delta(\underset{\sim}{\sigma}, \ell) \leq \max(m, n) < \infty$.*

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

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

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

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

Lemma 4.9. *The complexity A is upper bounded by $\mathcal{O}\left(\sum_{c=1}^{q} \binom{cn+n+c}{c} (|\Sigma| + 1)^c\right)$.*

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

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

[4] This reflects our general approach, which builds a surjection from the interleaved Hamming ball onto the Levenshtein ball.

4.8 Sampling the Levenshtein ball without replacement in $\mathcal{O}(1)$

Now that we have a reliable method to synthesize admissible completions for strings containing holes, i.e., fix *localized* errors, $F : (\mathcal{G} \times \underline{\Sigma}^n) \rightarrow \{\Sigma^n\} \subseteq \mathcal{L}(\mathcal{G})$, how can we use F to repair some unparseable string, i.e., $\underset{\sim}{\sigma_1 \ldots \sigma_n} : \Sigma^n \cap \mathcal{L}(\mathcal{G})^\complement$ where the holes' locations are unknown? Three questions stand out in particular: how many holes are needed to repair the string, where should we put those holes, and how ought we fill them to obtain a parseable $\tilde{\sigma} \in \mathcal{L}(\mathcal{G})$?

One plausible approach would be to draw samples with a PCFG, minimizing tree-edit distance, however these are computationally expensive metrics and approximations may converge poorly. A more efficient strategy is to sample string perturbations, $\boldsymbol{\sigma} \sim \Sigma^{n \pm q} \cap \Delta_q(\underset{\sim}{\sigma})$ uniformly across the Levenshtein q-ball centered on $\underset{\sim}{\sigma}$, i.e., the space of all admissible edits with Levenshtein distance $\leq q$.

To implement this strategy, we first construct a surjection $\varphi^{-1} : \mathbb{Z}_2^m \twoheadrightarrow \Delta_q(\underset{\sim}{\sigma})$ from bitvectors to Levenshtein edits over $\underset{\sim}{\sigma}, \Sigma$, sample bitvectors without replacement using a characteristic polynomial, then decode the resulting bitvectors into Levenshtein edits. This ensures the sampler eventually visits every Levenshtein edit at least exactly once and at most approximately once, without needing to store any samples, and discovers a steady stream of admissible edits throughout the solving process, independent of the grammar or string under repair.

More specifically, we employ a pair of [un]tupling functions $\kappa, \rho : \mathbb{N}^k \leftrightarrow \mathbb{N}$ which are (1) bijective (2) maximally compact (3) computationally tractable (i.e., closed form inverses). κ will be used to index $\left\{ {n \atop k} \right\}$[2]-combinations and ρ will index Σ^k tuples, but is slightly more tricky to define. To maximize compactness, there is an elegant pairing function by Szudzik [8], which enumerates concentric square shells over $\mathbb{N}^2$ and can be generalized to hypercubic shells in $\mathbb{N}^k$.

Although $\langle \kappa, \rho \rangle$ could be used directly to exhaustively search the Levenshtein ball, they are temporally biased samplers due to lexicographic ordering. Rather, we would prefer a path that uniformly visits every fertile subspace of the Levenshtein ball over time regardless of the grammar or string in question: subsequences of $\langle \kappa, \rho \rangle$ should discover valid repairs with frequency roughly proportional to the filtration rate, i.e., the density of the admissible set relative to the Levenshtein ball. These additional constraints give rise to two more criteria: (4) ergodicity and (5) periodicity.

To achieve ergodicity, we permute the elements of $\left\{ {n \atop k} \right\} \times \Sigma^k$ using a finite field with a characteristic polynomial C of degree $m := \lceil \log_b \binom{n}{k} |\Sigma_\varepsilon|^k \rceil$. By choosing C to be some irreducible polynomial, one ensures the path has the mixing properties we desire, e.g., suppose $U : \mathbb{Z}_2^{m \times m}$ is a matrix whose structure is depicted to the right, wherein C represents a primitive polynomial over $\mathbb{Z}_2^m$ with coefficients $C_{1 \ldots m}$ and semiring operators $\oplus := + \pmod 2, \otimes := \wedge, \top := 1, \circ := 0$. Since C is primitive, the sequence $\mathbf{R} = (U^{0 \ldots 2^m - 1} Y)$ must have *full periodicity*, i.e., for all $i, j \in [0, 2^m)$, $\mathbf{R}_i = \mathbf{R}_j \Rightarrow i = j$. To uniformly sample $\boldsymbol{\sigma}$ without replacement, we construct a partial surjection from $\mathbb{Z}_2^m$ onto the Levenshtein ball, $\mathbb{Z}_2^m \rightharpoonup \left\{ {n \atop d} \right\} \times \Sigma_\varepsilon^d$, cycle over $\mathbf{R}$, then discard samples which have no witness in $\left\{ {n \atop d} \right\} \times \Sigma_\varepsilon^d$.

$$U^\intercal Y = \begin{pmatrix} C_1 & \cdots & \cdots & \cdots & C_m \\ \top & \circ & \cdots & \cdots & \circ \\ \circ & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ \circ & \cdots & \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} Y_1 \\ \vdots \\ \vdots \\ \vdots \\ Y_m \end{pmatrix}$$

This procedure requires $\mathcal{O}(1)$ per sample and roughly $\binom{n}{d}|\Sigma_\varepsilon|^d$ samples to exhaustively search $\left\{ {n \atop d} \right\} \times \Sigma_\varepsilon^d$. Its acceptance rate $b^{-m}\binom{n}{d}|\Sigma_\varepsilon|^d$ can be slightly improved with a more suitable base b, however this introduces some additional complexity and so we elected to defer this optimization.

In addition to its statistically desirable properties, our sampler has the practical benefit of being trivially parallelizable using leapfrogging, i.e., given p independent processors, each one p_j can independently check $[\varphi^{-1}(\langle \kappa, \rho \rangle^{-1}(\mathbf{R}_i), \underset{\sim}{\sigma}) \in \mathcal{L}(\mathcal{G})]$ where $p_j \equiv i \pmod{|p|}$. This procedure

[2] Following Stirling, we use $\left\{ {n \atop d} \right\}$ to denote the set of all d-element subsets of $\{1, \ldots, n\}$.

linearly scales with the total processors, exhaustively searching $\Delta_q(\utilde{\sigma})$ in $|p|^{-1}$ of the time required by a single processor, or alternately drawing $|p|$ times as many samples in the same time. Although complete with respect to $\Delta_q(\utilde{\sigma})$, this approach can produce patches containing more Levenshtein edits than are strictly necessary to repair $\utilde{\sigma}$. To ensure patches are both minimal and syntactically valid, we first introduce a simple technique to minimize the repairs in §4.9. By itself, uniformly sampling minimal repairs $\tilde{\sigma} \sim \Delta_q(\utilde{\sigma}) \cap \mathcal{L}(\mathcal{G})$ is sufficient but can be quite time-consuming. To further reduce sample complexity and enable real-time repairs, we will then introduce a more efficient density estimator based on adaptive resampling (§4.10).

4.9 Patch minimization

Suppose we have a string, `a ( b`, and discover the patch, $\tilde{\sigma}$ = `( a + b )`. Although $\tilde{\sigma}$ is syntactically admissible, it is not minimal. To minimize a patch, we consider the set of all of its constituent subpatches, namely, `( a + b`, `( a ( b )`, `a + b )`, `( a ( b`, `a + b`, and `a ( b )`, then retain only the smallest syntactically valid instance(s) by Levenshtein distance. This forms a so-called *patch powerset*, which can be lazily enumerated from the top-down, after which we take all valid strings from the lowest level containing at least one valid string, i.e., `a + b` and `a ( b )`. When patches are very large, minimization can be used in tandem with the delta debugging technique [10] to first simplify contiguous edits, then apply the patch powerset construction. Minimization is often useful for estimating the language edit distance: given a single valid repair of arbitrary size, minimization lets us quickly approximate an upper-bound on $\Delta(\utilde{\sigma}, \ell)$.

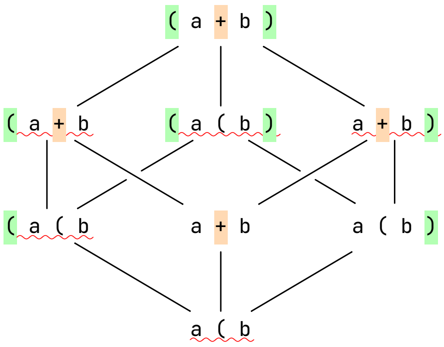


Fig. 3. The patch $\tilde{\sigma}$ = `( a + b )` is decomposed into its constituents.

4.10 Probabilistic reachability

Since there are $\Sigma_{d=1}^{q}\binom{n}{d}$ total hole templates, each with $|\Sigma_\varepsilon|^d$ individual edits to check, if n and q are large, this space can be slow to exhaustively search and a uniform prior may be highly sample-inefficient. Furthermore, naïvely sampling $\sigma \sim \Delta_q(\utilde{\sigma})$ is likely to produce a large number of unnatural edits and converge poorly on $\Delta_q(\utilde{\sigma}) \cap \mathcal{L}(\mathcal{G})$. To rapidly rank and render relevant repair recommendations, we prioritize candidate edits according to the following procedure.

(1) Draw samples $\hat{\sigma} \sim \Delta_q(\utilde{\sigma})$ without replacement using §4.8 with leapfrog parallelization. (2) Score by perplexity $PP(\hat{\sigma})$ using a pretrained variable-order Markov chain (VOMC) [7]. (3) Resample using a concurrent variant of the A-Res [4] online weighted reservoir sampler. (4) Filter Levenshtein edits by admissibility with respect to the grammar, i.e., $[\hat{\sigma} \in \mathcal{L}(\mathcal{G})]$. (5) Minimize and store admissible repairs to a replay buffer, $\mathcal{Q} \leftarrow \tilde{\sigma}$, ranked by perplexity. (6) Repeat steps (1)-(5), alternately sampling from the LFSR/VOMC-reweighted online resevoir sampler with probability ϵ or stochastically resampled $\mathcal{Q}$ with probability $(1 - \epsilon)$, where ϵ decreases from 1 to 0 according to a stepwise schedule relative to the time remaining.

Initially, the replay buffer $\mathcal{Q}$ is empty and repairs are sampled uniformly without replacement from the Levenshtein ball, $\Delta_q(\utilde{\sigma})$. As time progresses, $\mathcal{Q}$ is gradually populated with admissible repairs and resampled with increasing probability, allowing the algorithm to initially explore, then exploit the most promising candidates. This is summarized in Algorithm 1 which is run in parallel across all available CPU cores.

Algorithm 1 Probabilistic reachability

```
Require: $\mathcal{G}$ grammar, $\utilde{\sigma}$ broken string, $p$ process ID, $c$ total CPU cores, $t_{\text{total}}$ timeout.
Initialize replay buffer $\mathcal{Q} \leftarrow \varnothing$, reservoir $\mathcal{R} \leftarrow \varnothing$, $\epsilon \leftarrow 1, i \leftarrow 0, Y \sim \mathbb{Z}_2^m, t_0 \leftarrow t_{\text{now}}$
repeat
    if $\mathcal{Q} = \varnothing$ or Rand$(0, 1) < \epsilon$ then
        $\hat{\sigma} \leftarrow \varphi^{-1}\left(\langle\kappa, \rho\rangle^{-1}(U^{ci+p}Y), \utilde{\sigma}\right), i \leftarrow i + 1$          ▷ Sample WoR using LFSR.
    else
        $\hat{\sigma} \sim \mathcal{Q} +$ Noise$(\mathcal{Q})$          ▷ Sample replay buffer with additive noise.
    end if
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{\hat{\sigma}\}$          ▷ Insert repair candidate $\hat{\sigma}$ into reservoir $\mathcal{R}$.
    if $\mathcal{R}$ is full then
        $\hat{\sigma} \leftarrow \operatorname{argmin}_{\hat{\sigma} \in \mathcal{R}} PP(\hat{\sigma})$          ▷ Select lowest perplexity repair candidate.
        if $\hat{\sigma} \in \mathcal{L}(\mathcal{G})$ then
            $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\hat{\sigma}\}$          ▷ Insert successful repair into replay buffer.
        end if
        $\mathcal{R} \leftarrow \mathcal{R} \setminus \{\hat{\sigma}\}$          ▷ Remove checked sample from the reservoir.
    end if
    $\epsilon \leftarrow$ Schedule$\left((t_{\text{now}} - t_0)/t_{\text{total}}\right)$          ▷ Update exploration/exploitation rate.
until $t_{\text{total}}$ elapses.
return the lowest $\tilde{\sigma} \in \mathcal{Q}$ ranked by $PP(\tilde{\sigma})$.
```

We would prefer hole templates likely to yield repairs that are (1) admissible (i.e., grammatically correct) and (2) plausible (i.e., likely to have been written by a human author). To do so, we draw holes and rank admissible repairs using a probabilistic distance metric over $\Delta_q(\utilde{\sigma})$. For example, suppose we are given an invalid string, $\utilde{\sigma}_\varepsilon : \Sigma^{90}$ and $\mathcal{Q} \subseteq [0, |\sigma_\varepsilon|) \times \Sigma_\varepsilon^q$, a distribution over previously successful edits, which we can use to localize admissible repairs. Marginalizing onto $\utilde{\sigma}_\varepsilon$, the distribution $\mathcal{Q}(\utilde{\sigma}_\varepsilon)$ may take the form shown in Fig. 4.

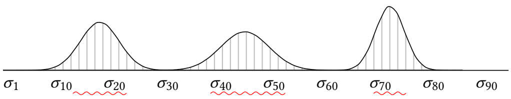
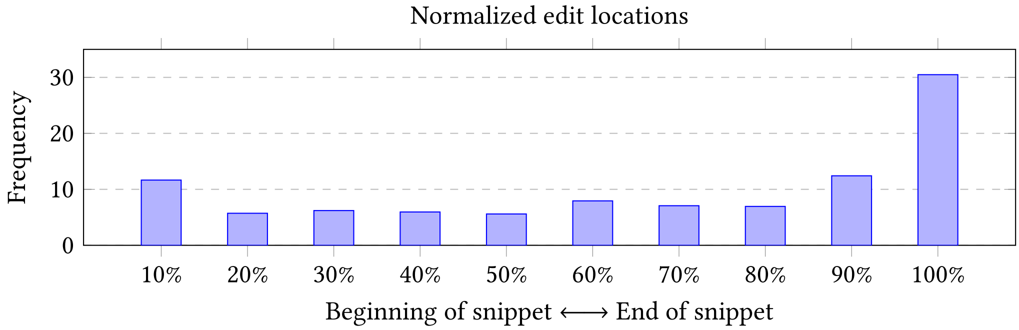


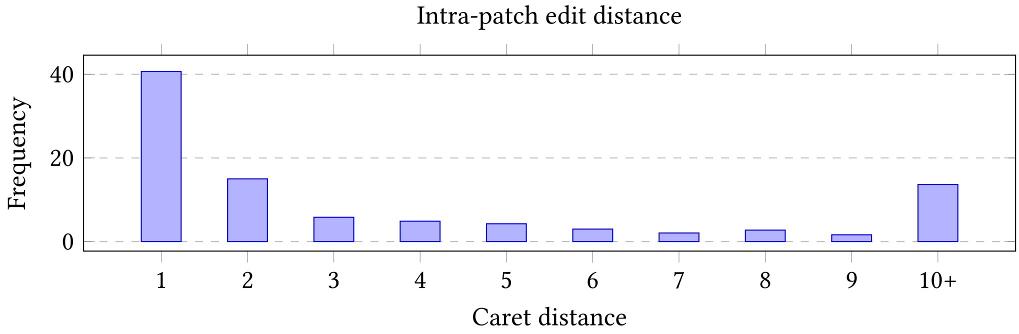
Fig. 4. The distribution $\mathcal{Q}$, projected onto σ, suggests edit locations likely to yield admissible repairs, from which we draw subsets of size d.

More specifically, we want to sample from a discrete product space that factorizes into (1) the edit locations (e.g., informed by caret position, historical edit locations, etc.), (2) probable completions (e.g., from a Markov chain or neural language model) and (3) an accompanying *cost model*, $C : (\Sigma^* \times \Sigma^*) \rightarrow \mathbb{R}$, which may be any number of suitable distance metrics, such as language edit distance, weighted Levenshtein distance, or stochastic contextual edit distance [3] in the case of probabilistic edits. Our goal then, is to discover repairs minimizing $C(\utilde{\sigma}, \tilde{\sigma})$, subject to the given grammar and latency constraints.

4.11 Trajectory Matching

Suppose we have a dataset of single token edits and their local context. For simplicity, we shall assume a trigram language model, i.e., $P(\sigma_i' \mid \sigma_{i-1}, \sigma_i, \sigma_{i+1})$, however the approach can be generalized to higher-order Markov models. Given a string σ, we can sample edit trajectories $q^1(\sigma), q^2(\sigma), \ldots, q^n(\sigma)$ by defining $q(\sigma)$ to sample a single edit from the set of all relevant edit actions $Q(\sigma)$, then recursively applying q to the resulting string. More formally,





1. Given a string σ, compute $Q(\sigma)$, the set of all relevant edit actions for all possible edit locations by unioning the set of all possible edits at each location, i.e., $Q(\sigma) \coloneqq \bigcup_{i=1}^{|\sigma|-1} \left\{\sigma_i' \mid 0 < P(\sigma_i \mid \sigma_{i-1}, \sigma_i, \sigma_{i+1})\right\}$.
2. Renormalize the probabilities of each edit $P(q \mid \sigma)$ by $\sum_{q \in Q(\sigma)} P(q)$. This ensures the probability of sampling a particular edit is proportional to its relative probability under the language model and sums to 1.
3. Sample an edit $q(\sigma) \sim Q(\sigma)$, then repeat for n steps where n is sampled from a geometric distribution with mean μ matching the average edit distance of the dataset (this assumes the edit distance is independent of the edits).

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 EVALUATION

For our evaluation, we use the StackOverflow dataset from [5]. 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. 5. 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.

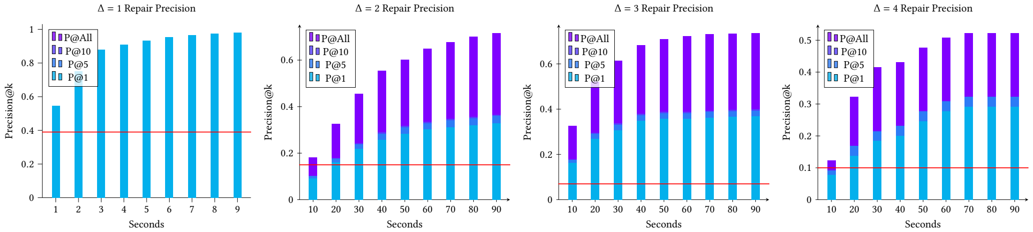


Fig. 6. Human repair benchmark. Note the y-axis across different edit distance plots has varying ranges.

6.1 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

6.2 Uniform sampling benchmark

Below, we plot the precision of the uniform sampling procedure described in §4.8 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.

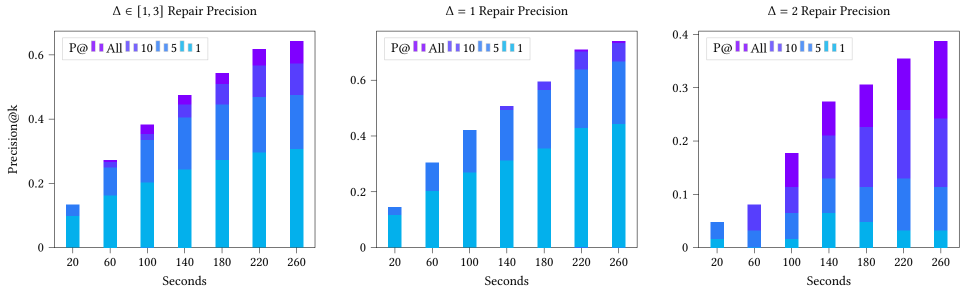


Fig. 7. Human repair benchmark. Note the y-axis across different edit distance plots has varying ranges.

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.

6.3 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 §4.10. 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.

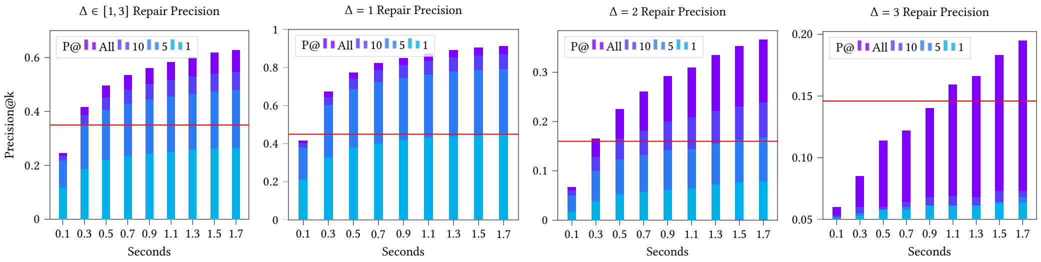


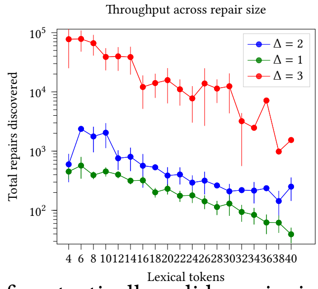
Fig. 8. Adaptive sampling repairs. The red line indicates Seq2Parse precision@1 on the same dataset. Since it only supports generating one repair, we do not report precision@k or the intermediate latency cutoffs.

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 **??**.

6.4 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.



6.5 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.

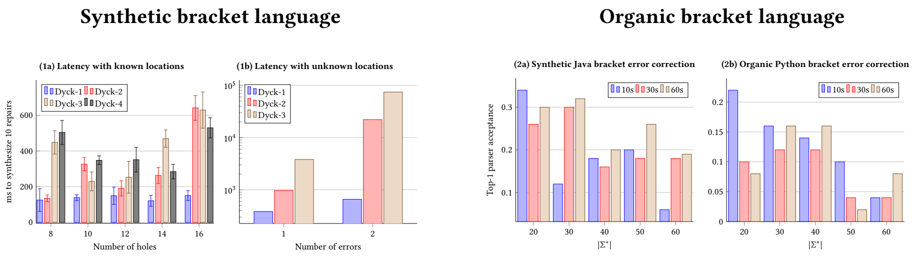


Fig. 9. Benchmarking bracket correction latency and accuracy across two bracketing languages, one generated from Dyck-n, and the second uses an abstracted source code snippet with imbalanced parentheses.

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[5], 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 [9], 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.

REFERENCES

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

[2] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. 2019. Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. SIAM J. Comput. 48, 2 (2019), 481–512.

[3] Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2014. Stochastic Contextual Edit Distance and Probabilistic FSTs. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, Vol. 2 (Short Papers). Association for Computational Linguistics, Baltimore, Maryland, 625–630.

[4] Pavlos S Efraimidis. 2015. Weighted random sampling over data streams. Algorithms, Probability, Networks, and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday (2015), 183–195.

[5] 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

[6] Rohit J. Parikh. 1966. On Context-Free Languages. J. ACM 13, 4 (oct 1966), 570–581. https://doi.org/10.1145/321356.321364

[7] Marcel H Schulz, David Weese, Tobias Rausch, Andreas Döring, Knut Reinert, and Martin Vingron. 2008. Fast and adaptive variable order Markov chain construction. In Algorithms in Bioinformatics: 8th International Workshop, WABI 2008, Karlsruhe, Germany, September 15-19, 2008. Proceedings 8. Springer, 306–317.

[8] Matthew Szudzik. 2006. An elegant pairing function. In Special NKS 2006 Wolfram Science Conference. 1–12.

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

[10] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. ACM SIGSOFT Software Engineering Notes 27, 6 (2002), 1–10.

[5] 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.