

Repairing Multiline Syntax Errors Using the Levenshtein-Bar-Hillel Construction

Breandan Considine, Jin Guo, Xujie Si

McGill University, Mila IQIA

bre@ndan.co

November 17, 2024



Overview

- 1 Formal Language Theory
- 2 Algebraic Parsing
- 3 Decoding

Can you spot the error?

Original code	Human repair
<pre>newlist = [] i = set([5, 3, 1]) z = set([5, 0, 4])</pre>	

Can you spot the error?

Original code	Human repair
<pre>newlist = [] i = set([5, 3, 1]) z = set([5, 0, 4])</pre>	<pre>newlist = [] i = set([5, 3, 1]) z = set([5, 0, 4])</pre>

Can you spot the error?

Original code	Human repair
<pre>def average(values): if values = (1,2,3): return (1+2+3)/3 else if values = (-3,2): return (-3+2+8-1)/4</pre>	

Can you spot the error?

Original code	Human repair
<pre>def average(values): if values == (1,2,3): return (1+2+3)/3 else if values == (-3,2): return (-3+2+8-1)/4</pre>	<pre>def average(values): if values == (1,2,3): return (1+2+3)/3 elif values == (-3,2): return (-3+2+8-1)/4</pre>

Can you spot the error?

Original code	Human repair
<pre>import Global from Global globalObj = Global() print(str(globalObj.Test()))</pre>	

Can you spot the error?

Original code	Human repair
<pre>import Global from Global globalObj = Global() print(str(globalObj.Test()))</pre>	<pre>from Global import Global globalObj = Global() print(str(globalObj.Test()))</pre>

How many repairs could there possibly be?

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)]
```

How many repairs could there possibly be?

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)]
```

It can be fixed by appending a colon after the function signature, yielding:

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

How many repairs could there possibly be?

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)]
```

It can be fixed by appending a colon after the function signature, yielding:

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

Let us consider a slightly more ambiguous error: `v = df.iloc(5:, 2:)`. Assuming an alphabet of just a hundred lexical tokens, this statement has millions of two-token edits, yet only six 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])`

Parsing for linear algebraists

Given a CFG $\mathcal{G} := \langle V, \Sigma, P, S \rangle$ in Chomsky Normal Form, we can construct a recognizer $R_{\mathcal{G}} : \Sigma^n \rightarrow \mathbb{B}$ for strings $\sigma : \Sigma^n$ as follows. Let 2^V be our domain, 0 be \emptyset , \oplus be \cup , and \otimes be defined as follows:

$$s_1 \otimes s_2 := \{C \mid \langle A, B \rangle \in s_1 \times s_2, (C \rightarrow AB) \in P\}$$

e.g., $\{A \rightarrow BC, C \rightarrow AD, D \rightarrow BA\} \subseteq P \vdash \{A, B, C\} \otimes \{B, C, D\} = \{A, C\}$

If we define $\sigma_r^\dagger := \{w \mid (w \rightarrow \sigma_r) \in P\}$, then initialize

$M_{r+1=c}^0(\mathcal{G}', e) := \sigma_r^\dagger$ and solve for the fixpoint $M^* = M + M^2$,

$$M^0 := \begin{pmatrix} \emptyset & \sigma_1^\rightarrow & \emptyset & \dots & \emptyset \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \emptyset & \dots & \emptyset & \dots & \emptyset \end{pmatrix} \Rightarrow \dots \Rightarrow M^* = \begin{pmatrix} \emptyset & \sigma_1^\rightarrow & \Lambda & \dots & \Lambda_\sigma^* \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \emptyset & \dots & \emptyset & \dots & \emptyset \end{pmatrix}$$

$$S \Rightarrow^* \sigma \iff \sigma \in \mathcal{L}(\mathcal{G}) \text{ iff } S \in \Lambda_\sigma^*, \text{ i.e., } \mathbb{1}_{\Lambda_\sigma^*}(S) \iff \mathbb{1}_{\mathcal{L}(\mathcal{G})}(\sigma).$$

Satisfiability + holes

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 $\oplus = \cup$, $X \otimes Z = \{w \mid \langle x, z \rangle \in X \times Z, (w \rightarrow xz) \in P\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

$$M_0: \begin{array}{c} \begin{array}{c} \overline{2^V} \\ \left(\begin{array}{ccc} \{N\} & & \\ & \{N, O\} & \\ & & \{N, O\} \end{array} \right) \end{array} \quad \begin{array}{c} \overline{\mathbb{Z}_2^{|V|}} \\ \left(\begin{array}{ccc} \square \blacksquare \square \square & & \\ & \square \blacksquare \blacksquare \square & \\ & & \square \blacksquare \blacksquare \square \end{array} \right) \end{array} \quad \begin{array}{c} \overline{\mathbb{Z}_2^{|V|} \rightarrow \mathbb{Z}_2^{|V|}} \\ \left(\begin{array}{ccc} V_{0,1} & & \\ & V_{1,2} & \\ & & V_{2,3} \end{array} \right) \end{array} \end{array}$$

Satisfiability + holes

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 $\oplus = \cup$, $X \otimes Z = \{w \mid \langle x, z \rangle \in X \times Z, (w \rightarrow xz) \in P\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

	2^V	$\mathbb{Z}_2^{ V }$	$\mathbb{Z}_2^{ V } \rightarrow \mathbb{Z}_2^{ V }$
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 :	$\begin{pmatrix} \{N\} & \emptyset & \\ \{N, O\} & \{L\} & \\ \{N, O\} & \{N, O\} & \end{pmatrix}$	$\begin{pmatrix} \square \blacksquare \square \square & \square \square \square \square & \\ \square \blacksquare \blacksquare \square & \blacksquare \square \square \square & \\ \square \blacksquare \blacksquare \square & \square \blacksquare \blacksquare \square & \end{pmatrix}$	$\begin{pmatrix} V_{0,1} & V_{0,2} & \\ V_{1,2} & V_{1,3} & \\ V_{2,3} & & \end{pmatrix}$

Satisfiability + holes

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 $\oplus = \cup$, $X \otimes Z = \{w \mid \langle x, z \rangle \in X \times Z, (w \rightarrow xz) \in P\}$, the fixpoint $M' = M + M^2$ can be computed as follows:

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

Satisfiability + holes

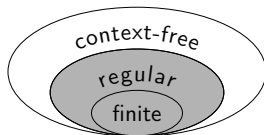
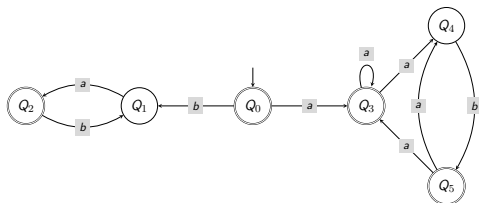
Let us consider an example with two holes, $\sigma = 1 _ _$, and the grammar being $G := \{S \rightarrow NON, O \rightarrow + \mid \times, N \rightarrow 0 \mid 1\}$. This can be rewritten into CNF as $G' := \{S \rightarrow NL, N \rightarrow 0 \mid 1, O \rightarrow \times \mid +, L \rightarrow ON\}$.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

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

Background: Regular grammars

A regular grammar (RG) is a quadruple $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ where V are nonterminals, Σ are terminals, $P : V \times (V \cup \Sigma)^{\leq 2}$ are the productions, and $S \in V$ is the start symbol, i.e., all productions are of the form $A \rightarrow a$, $A \rightarrow aB$ (right-regular), or $A \rightarrow Ba$ (left-regular). e.g., the following RG and NFA correspond to the language defined by the *regex* $(a(ab)^*)^*(ba)^*$:

$S \rightarrow Q_0 \mid Q_2 \mid Q_3 \mid Q_5$
 $Q_0 \rightarrow \varepsilon$
 $Q_1 \rightarrow Q_0b \mid Q_2b$
 $Q_2 \rightarrow Q_1a$
 $Q_3 \rightarrow Q_0a \mid Q_3a \mid Q_5a$
 $Q_4 \rightarrow Q_3a \mid Q_5a$
 $Q_5 \rightarrow Q_4b$



Levenshtein automaton customization

Consider the string $\sigma = ())$ and the alphabet $\Sigma = \{), (\}$. Every string within one edit of σ is recognized by an NFA with the following structure:

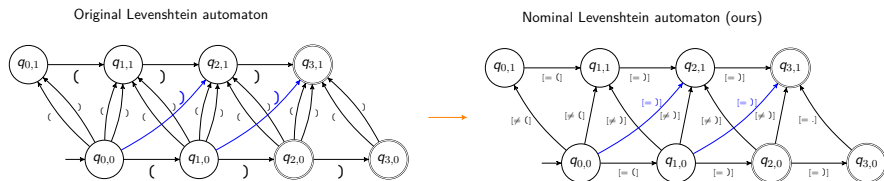


Figure: Automaton recognizing every single patch of the broken string $())$ within Levenshtein distance 1. We nominalize the original Levenshtein automaton, ensuring upward arcs denote a mutation, and replace terminals with a symbolic predicate, which deduplicates parallel arcs in large alphabets.

<https://fulmicoton.com/posts/levenshtein/#observations-lets-count-states>

Levenshtein reachability

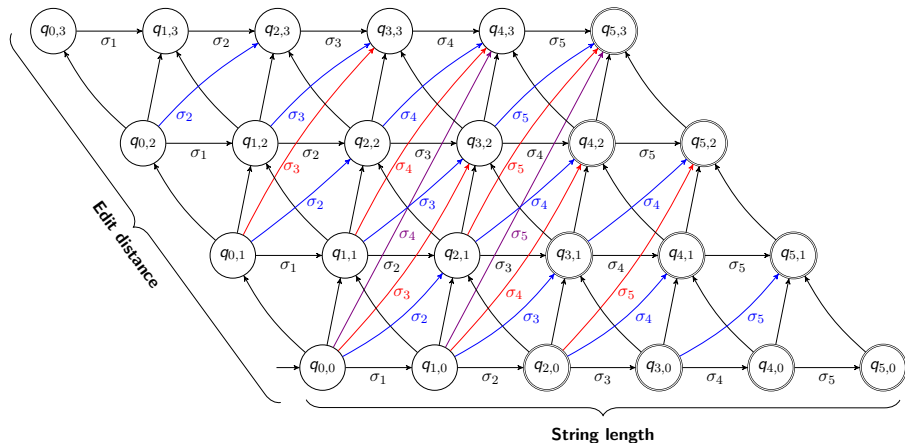


Figure: Bounded Levenshtein reachability from $\sigma : \Sigma^n$ is expressible as an NFA populated by accept states within radius k of $S = q_{n,0}$, which accepts all strings σ' within Levenshtein radius k of σ .

The nominal Levenshtein automaton

The original Levenshtein automaton (Schulz & Stoyan, 2002):

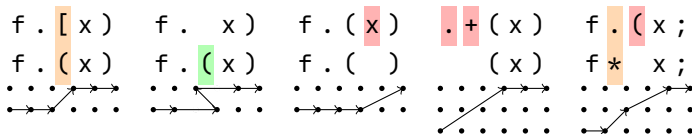
$$\begin{array}{c}
 \frac{s \in \Sigma \quad i \in [0, n] \quad j \in [1, k]}{(q_{i,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \uparrow \qquad \frac{s \in \Sigma \quad i \in [1, n] \quad j \in [1, k]}{(q_{i-1,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nearrow \\
 \frac{s = \sigma_i \quad i \in [1, n] \quad j \in [0, k]}{(q_{i-1,j} \xrightarrow{s} q_{i,j}) \in \delta} \rightarrow \qquad \frac{s = \sigma_i \quad i \in [2, n] \quad j \in [1, k]}{(q_{i-2,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nearrow \\
 \frac{}{q_{0,0} \in I} \text{INIT} \qquad \frac{q_{i,j} \quad |n - i + j| \leq k}{q_{i,j} \in F} \text{DONE}
 \end{array}$$

We modify the original automaton with a nominal predicate:

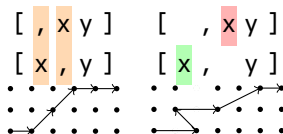
$$\begin{array}{c}
 \frac{i \in [0, n] \quad j \in [1, k]}{(q_{i,j-1} \xrightarrow{[\neq \sigma_{i+1}]} q_{i,j}) \in \delta} \uparrow \qquad \frac{i \in [1, n] \quad j \in [1, k]}{(q_{i-1,j-1} \xrightarrow{[\neq \sigma_i]} q_{i,j}) \in \delta} \nearrow \\
 \frac{i \in [1, n] \quad j \in [0, k]}{(q_{i-1,j} \xrightarrow{[= \sigma_i]} q_{i,j}) \in \delta} \rightarrow \qquad \frac{d \in [1, d_{\max}] \quad i \in [d+1, n] \quad j \in [d, k]}{(q_{i-d-1,j-d} \xrightarrow{[= \sigma_i]} q_{i,j}) \in \delta} \nearrow
 \end{array}$$

Geometrically interpreting the edit calculus

Each arc plays a specific role. \uparrow handles insertions, \nearrow handles substitutions and \nearrow handles deletions of ≥ 1 tokens. Consider some illustrative cases:



Note that the same $\langle \sigma, \sigma' \rangle$ pair can have multiple Levenshtein alignments:



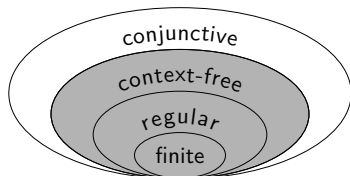
Non-uniqueness of geodesics has implications for $\text{CFG} \cap \text{L-NFA}$ ambiguity.

Background: Context-free grammars

In a context-free grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ all productions are of the form $P : V \times (V \cup \Sigma)^+$, i.e., RHS may contain any number of nonterminals, V . Recognition decidable in $\mathcal{O}(n^\omega)$, n.b. CFLs are **not** closed under \cap !

For example, consider the grammar $S \rightarrow SS \mid (S) \mid ()$. This represents the language of balanced parentheses, e.g. $()$, $()()$, $(())$, $()(())$, $(())()$, $(())()$...

Every CFG has a normal form $P^* : V \times (V^2 \mid \Sigma)$, i.e., every production can be refactored into either $v_0 \rightarrow v_1 v_2$ or $v_0 \rightarrow \sigma$, where $v_{0...2} : V$ and $\sigma : \Sigma$, e.g., $\{S \rightarrow SS \mid (S) \mid ()\} \Leftrightarrow^* \{S \rightarrow XR \mid SS \mid LR, L \rightarrow (, R \rightarrow), X \rightarrow LS\}$



Background: Closure properties of formal languages

Formal languages are not always closed under set-theoretic operations, e.g., $\text{CFL} \cap \text{CFL}$ is not CFL in general. Let \cdot denote concatenation, $*$ be Kleene star, and \complement be complementation:

	\cup	\cap	\cdot	$*$	\complement
Finite ¹	✓	✓	✓	✓	✓
Regular ¹	✓	✓	✓	✓	✓
Context-free ^{1,2}	✓	✗ [†]	✓	✓	✗
Conjunctive ^{1,2}	✓	✓	✓	✓	?
Context-sensitive ²	✓	✓	✓	+	✓
Recursively Enumerable ²	✓	✓	✓	✓	✗

We would like a language family that is (1) tractable, i.e., has polynomial recognition and search complexity and (2) reasonably expressive, i.e., can represent syntactic properties of real-world programming languages.

[†] However, CFLs are closed under intersection with regular languages.

The Bar-Hillel construction and its specialization

Theorem (Bar-Hillel, 1961)

Given a CFG, G , and an NFA, A , there exists a $G_{\cap} = \langle V_{\cap}, \Sigma_{\cap}, P_{\cap}, S \rangle$, such that $\mathcal{L}(G_{\cap}) = \mathcal{L}(G) \cap \mathcal{L}(A)$.

The Bar-Hillel construction and its specialization

Theorem (Bar-Hillel, 1961)

Given a CFG, G , and an NFA, A , there exists a $G_{\cap} = \langle V_{\cap}, \Sigma_{\cap}, P_{\cap}, S \rangle$, such that $\mathcal{L}(G_{\cap}) = \mathcal{L}(G) \cap \mathcal{L}(A)$.

Salomaa (1973) constructs the intersection grammar as follows:

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

The Bar-Hillel construction and its specialization

Theorem (Bar-Hillel, 1961)

Given a CFG, G , and an NFA, A , there exists a $G_{\cap} = \langle V_{\cap}, \Sigma_{\cap}, P_{\cap}, S \rangle$, such that $\mathcal{L}(G_{\cap}) = \mathcal{L}(G) \cap \mathcal{L}(A)$.

Salomaa (1973) constructs the intersection grammar as follows:

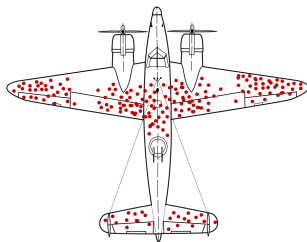
$$\frac{q \in I \quad r \in F}{(S \rightarrow qSr) \in P_{\cap}} \quad \checkmark \qquad \frac{(A \rightarrow a) \in P \quad (q \xrightarrow{a} r) \in \delta}{(qAr \rightarrow a) \in P_{\cap}} \quad \uparrow$$
$$\frac{(w \rightarrow xz) \in P \quad p, q, r \in Q}{(pwr \rightarrow (pxq)(qzr)) \in P_{\cap}}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

Observation: too many (q, v, q') triples! Three low-hanging optimizations:

- 1 Only consider (q, q') where $\exists \sigma : \Sigma^* \text{ s.t. } q \xrightarrow{\sigma} q'$.
- 2 Filter impossible (q, v, q') triples based on path length.
- 3 Remove unreachable states q' , i.e., $\nexists \sigma \in \mathcal{L}(G) \text{ s.t. } q_0 \xrightarrow{\sigma} q'$.

Edit location refinement: intuition

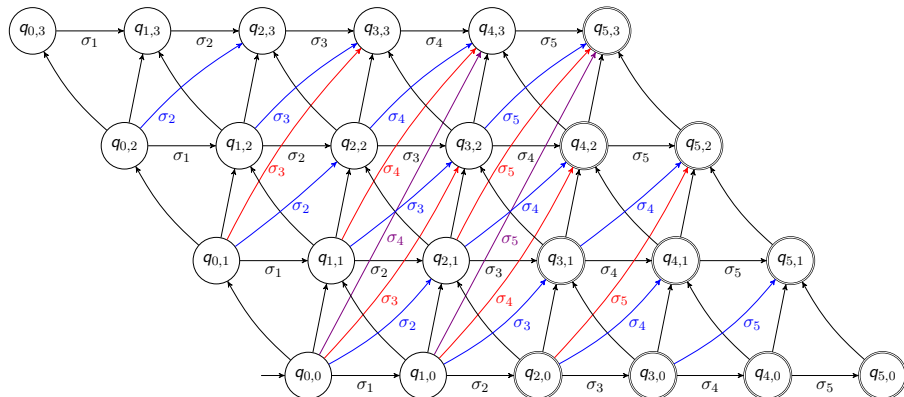
Let's think: do we really need $|\sigma| \times d_{\max}$ states? Can we somehow narrow down the edit range? Where can we cut down on armor?



It is typically easier to determine where *not* to make the edits. Certain regions, no matter their contents, will never yield a viable repair.

Edit location refinement: example

Let's prune states absorbing obviously impossible repair trajectories!

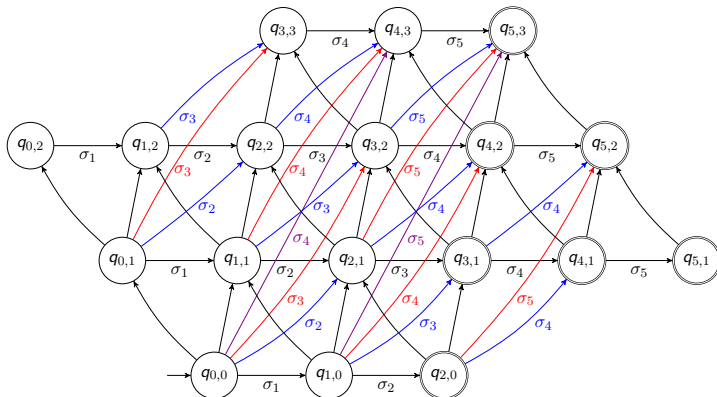


$$G: S \rightarrow \begin{pmatrix} S \\ S + S \end{pmatrix} \mid \begin{matrix} [S] \\ S + S \end{matrix} \mid 1$$

$$\sigma: [(+)] \quad \text{X}$$

Edit location refinement: example

Let's prune states absorbing obviously impossible repair trajectories!



$$G: S \rightarrow \begin{pmatrix} S \\ S \\ S + S \end{pmatrix} \mid \begin{matrix} 1 \\ 1 \\ 1 \end{matrix}$$

$$\sigma: \begin{bmatrix} (+) \\ - - +) \\ (+ - - \end{bmatrix} \begin{matrix} \text{Red X} \\ \text{Red X} \\ \text{Red X} \end{matrix} \wedge \begin{bmatrix} - - -) \\ (- - - \end{bmatrix} \begin{matrix} \text{Green Check} \\ \text{Green Check} \end{matrix}$$

Grammar refinement: Parikh interval maps

If we're reusing the CFG, it makes sense to precompute some statistics. For example, the total tokens each nonterminal can parse.

Definition (Parikh map for nonterminals)

The **Parikh image**, $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$, counts the number of times each terminal appears in a string. The **Parikh map**, $\pi : V \times \mathbb{N} \rightarrow \Pi$, is the smallest interval containing the Parikh image across all strings generated by a nonterminal R of length n , i.e., $\forall \sigma \in \mathcal{L}(R) \cap \Sigma^n \vdash p(\sigma) \in \pi(R, n)$.

Grammar refinement: Parikh interval maps

If we're reusing the CFG, it makes sense to precompute some statistics. For example, the total tokens each nonterminal can parse.

Definition (Parikh map for nonterminals)

The **Parikh image**, $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$, counts the number of times each terminal appears in a string. The **Parikh map**, $\pi : V \times \mathbb{N} \rightarrow \Pi$, is the smallest interval containing the Parikh image across all strings generated by a nonterminal R of length n , i.e., $\forall \sigma \in \mathcal{L}(R) \cap \Sigma^n \vdash p(\sigma) \in \pi(R, n)$.

Grammar refinement: Parikh interval maps

If we're reusing the CFG, it makes sense to precompute some statistics. For example, the total tokens each nonterminal can parse.

Definition (Parikh map for nonterminals)

The **Parikh image**, $p : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$, counts the number of times each terminal appears in a string. The **Parikh map**, $\pi : V \times \mathbb{N} \rightarrow \Pi$, is the smallest interval containing the Parikh image across all strings generated by a nonterminal R of length n , i.e., $\forall \sigma \in \mathcal{L}(R) \cap \Sigma^n \vdash p(\sigma) \in \pi(R, n)$.

Likewise, we can define a Parikh interval for subautomata.

Definition (Parikh map for subautomata)

The **Parikh map** for a subautomaton, $\pi : Q \times Q \rightarrow \Pi$, returns the smallest interval s.t. $\forall \sigma : \Sigma^*, q, q' : Q, q \xRightarrow{\sigma} q' \vdash p(\sigma) \in \pi(q, q')$.

Now when we have a new automaton, we can check whether (q, v, q') is compatible by checking whether the Parikh range for q, q' and v overlaps.

Grammar refinement: Parikh compatibility

Criteria to overapproximate compatible nonterminals and states:

Definition (Parikh compatibility)

We call a nonterminal R and state pair q, q' *compatible* when:

$$\frac{\mathcal{L}(R) \cap \bigcup_{n \in \llbracket q, q' \rrbracket} \Sigma^n \neq \emptyset \quad \pi(q, q') \cap \bigcup_{n \in \llbracket q, q' \rrbracket} \pi(R, n) \neq \emptyset}{R \triangleleft q, q'} \triangleleft$$

where $\llbracket q, q' \rrbracket = [\min |\sigma|, \max |\sigma|]$ s.t. $q \xrightarrow{\sigma} q'$.

Now, we are ready to re[de]fine the original \bowtie rule:

$$\frac{(w \rightarrow xz) \in P \quad p, q, r \in Q}{(pwr \rightarrow (pxq)(qzr)) \in P_{\cap}} \bowtie$$

Grammar refinement: Parikh compatibility

Criteria to overapproximate compatible nonterminals and states:

Definition (Parikh compatibility)

We call a nonterminal R and state pair q, q' *compatible* when:

$$\frac{\mathcal{L}(R) \cap \bigcup_{n \in \llbracket q, q' \rrbracket} \Sigma^n \neq \emptyset \quad \pi(q, q') \cap \bigcup_{n \in \llbracket q, q' \rrbracket} \pi(R, n) \neq \emptyset}{R \triangleleft q, q'} \triangleleft$$

where $\llbracket q, q' \rrbracket = [\min |\sigma|, \max |\sigma|]$ s.t. $q \xrightarrow{\sigma} q'$.

Now, we are ready to re[de]fine the original \bowtie rule:

$$\frac{w \triangleleft pr \quad x \triangleleft pq \quad z \triangleleft qr \quad (w \rightarrow xz) \in P \quad p, q, r \in Q}{(pwr \rightarrow (pxq)(qzr)) \in P_{\cap}} \hat{\bowtie}$$

So, we have a grammar. Now what?

There still many unnecessary productions, since we over-approximated the grammar. To reduce the number of productions, we normalize $G_{\cap} \Rightarrow G_{\cap}^*$ to CNF, removing unreachable and unproductive productions.

We could sample repairs from G_{\cap}^* directly, but there are a few issues here:

- ① It is not obvious how to steer a CFG sampler with a LLM.
- ② It is nontrivial to implement a parallel, replacement-free CFG sampler while also prioritizing high-likelihood repairs.
- ③ Even if we could, this would not guarantee repair uniqueness, because the CFG can be ambiguous, so we could end up sampling distinct trees representing the same string.

Now, let's step back and think...

So, we have a grammar. Now what?

There still many unnecessary productions, since we over-approximated the grammar. To reduce the number of productions, we normalize $G_{\cap} \Rightarrow G_{\cap}^*$ to CNF, removing unreachable and unproductive productions.

We could sample repairs from G_{\cap}^* directly, but there are a few issues here:

- 1 It is not obvious how to steer a CFG sampler with a LLM.
- 2 It is nontrivial to implement a parallel, replacement-free CFG sampler while also prioritizing high-likelihood repairs.
- 3 Even if we could, this would not guarantee repair uniqueness, because the CFG can be ambiguous, so we could end up sampling distinct trees representing the same string.

Now, let's step back and think...the intersection language is finite...

So, we have a grammar. Now what?

There still many unnecessary productions, since we over-approximated the grammar. To reduce the number of productions, we normalize $G_{\cap} \Rightarrow G_{\cap}^*$ to CNF, removing unreachable and unproductive productions.

We could sample repairs from G_{\cap}^* directly, but there are a few issues here:

- 1 It is not obvious how to steer a CFG sampler with a LLM.
- 2 It is nontrivial to implement a parallel, replacement-free CFG sampler while also prioritizing high-likelihood repairs.
- 3 Even if we could, this would not guarantee repair uniqueness, because the CFG can be ambiguous, so we could end up sampling distinct trees representing the same string.

Now, let's step back and think...the intersection language is finite...
So we should be able to represent it as a DFA.

Potential ambiguity of Levenshtein-Bar-Hillel grammars

The previous technique enumerates parse trees in a given \mathbb{T}_2 , but does not guarantee string uniqueness since the CFG may be ambiguous.

Lemma (Possible ambiguity of LBH grammar)

If the FSA, α , is ambiguous, the intersection CFG, G_\cap , can be ambiguous.

Proof.

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

We can eliminate ambiguity and thereby improve the rate of convergence for natural syntax repair by first translating \mathbb{T}_2 into an FSA.

Existence of an FSA that generates $\mathcal{L}(G_\cap)$

There is an FSA generating $\mathcal{L}(G_\cap)$. We first show this non-constructively:

Lemma (Acyclicity of LBH grammar)

The intersection grammar, G_\cap , is acyclic.

Proof.

Assume G_\cap is cyclic. Then $\mathcal{L}(G_\cap)$ must be infinite. But since G_\cap generates $\ell \cap \mathcal{L}(\alpha)$ by construction and α is acyclic, $\mathcal{L}(G_\cap)$ is necessarily finite. Therefore, G_\cap must not be cyclic. □

Since G_\cap is acyclic and thus finite, it must be representable as an FSA. Using an FSA for decoding has many advantages, notably, it can be efficiently minimized and decoded in order of n-gram likelihood using a Markov chain or standard pretrained autoregressive language model.

Translating from T_2 to a DFA

Let $+, * : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ be automata operators satisfying the property $\mathcal{L}(A_1 + A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and $\mathcal{L}(A_1 * A_2) = \mathcal{L}(A_1) \times \mathcal{L}(A_2)$. We can translate \mathbb{T}_2 to \mathcal{A} , as follows, recalling FSAs are closed over $+, *$:

$$\mathcal{Y}(T : \mathbb{T}_2) \mapsto \begin{cases} \alpha \mid \mathcal{L}(\alpha) = \{T\} & T : \Sigma, \\ \sum_{\langle T_1, T_2 \rangle \in \text{children}(T)} \mathcal{Y}(T_1) * \mathcal{Y}(T_2) & T : VL(V^2 P(a)^2) \end{cases}$$

In the case of LBH intersections, $\mathcal{Y}(G'_\cap)$ yields $\alpha : \mathcal{A} \mid \mathcal{L}(\alpha) = \ell \cap L(\sigma, d)$, which can be minimized via Brzozowski's algorithm then decoded:

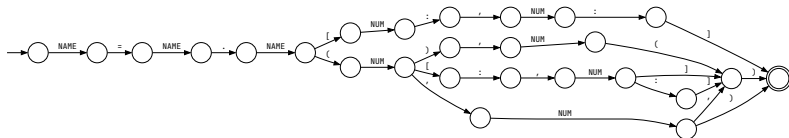


Figure: $L(\text{NAME} = \text{NAME} . \text{NAME} (\text{NUM} : , \text{NUM} :), 2) \cap \ell_{\text{PYTHON}}$

Decoding the DFA in order of normalized log likelihood

Algorithm Steerable DFA walk

Require: $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$ DFA, $P_\theta : \Sigma^d \rightarrow \mathbb{R}$ Markov chain

- 1: $\mathcal{T} \leftarrow \emptyset$ total trajectories, $\mathcal{P} \leftarrow [\langle \varepsilon, i, 0 \rangle \mid i \in I]$ partial trajectories
 - 2: **repeat**
 - 3: **let** $\langle \sigma, q, \gamma \rangle = \text{head}(\mathcal{P})$ **in**
 - 4: $\mathbf{T} = \{ \langle s\sigma, q', \gamma - \log P_\theta(s \mid \sigma_{1..d-1}) \rangle \mid (q \xrightarrow{s} q') \in \delta \}$
 - 5: **for** $\langle \sigma, q, \gamma \rangle = T \in \mathbf{T}$ **do**
 - 6: **if** $\exists s : \Sigma, q' : Q \mid (q \xrightarrow{s} q') \in \delta$ **then**
 - 7: $\mathcal{P} \leftarrow \text{tail}(\mathcal{P}) \oplus T$ \triangleright Add partial trajectory to PQ.
 - 8: **if** $q \in F$ **then**
 - 9: $\mathcal{T} \leftarrow \mathcal{T} \oplus T$ \triangleright Accepting state reached, add to queue.
 - 10: **until** interrupted or $\mathcal{P} = \emptyset$.
 - 11: **return** $[\sigma_{|\sigma|..1} \mid \langle \sigma, q, \gamma \rangle = T \in \mathcal{T}]$ \triangleright Return in sorted order
-

Characteristics of the repair dataset

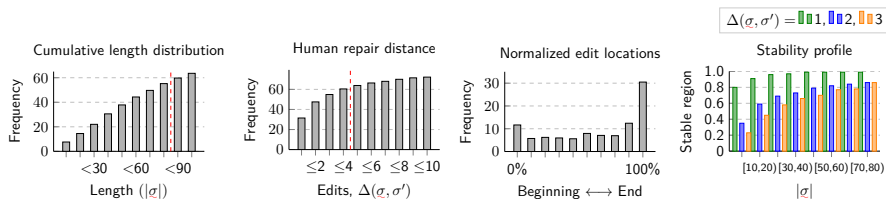


Figure: Repair statistics across the StackOverflow dataset, of which Tidyparse can handle about half in under ~ 30 s and ~ 150 GB. Larger repairs and edit distances are possible, albeit requiring additional time and memory. The stability profile measures the average fraction of all edit locations that were never altered by any repair in the $L(\sigma, \Delta(\sigma, \sigma'))$ -ball across repairs of varying length and distance.

Ranked repair

We train on lexical n-grams using the standard MLE for Markov chains. To score the repairs, we use the conventional length-normalized NLL:

$$\text{NLL}(\sigma) = -\frac{1}{|\sigma|} \sum_{i=1}^{|\sigma|} \log P_{\theta}(\sigma_i \mid \sigma_{i-1} \dots \sigma_{i-n}) \quad (1)$$

For each retrieved set $\hat{A} \subseteq A$ drawn before a predetermined timeout and each $\sigma \in \hat{A}$, we score the repair and return \hat{A} in ascending order.

To evaluate the quality of our ranking, we use the Precision@k statistic. Specifically, given a repair model, $R : \Sigma^* \rightarrow 2^{\Sigma^*}$ and a parallel corpus, $\mathcal{D}_{\text{test}}$, of errors (σ^{\dagger}) and repairs (σ'), we define Precision@k as:

$$\text{Precision@k}(R) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{\langle \sigma^{\dagger}, \sigma' \rangle \in \mathcal{D}_{\text{test}}} \mathbb{1} \left[\sigma' \in \underset{\sigma \subset R(\sigma^{\dagger}), |\sigma| \leq k}{\operatorname{argmax}} \sum_{\sigma \in \sigma} \text{NLL}(\sigma) \right] \quad (2)$$

Precision and latency comparison

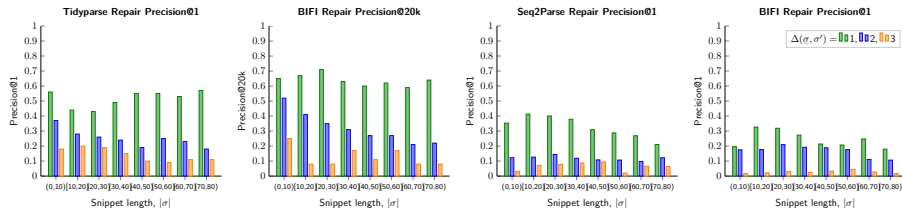


Figure: Tidyparse, Seq2Parse and BIFI repair precision across length and edits.

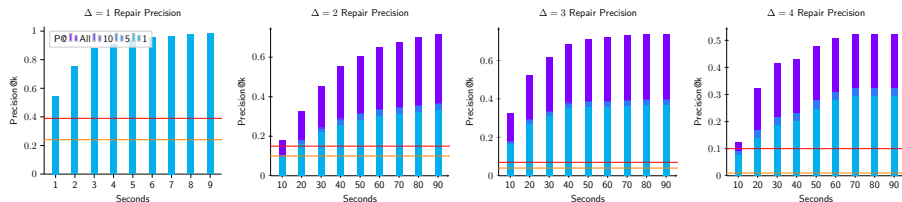
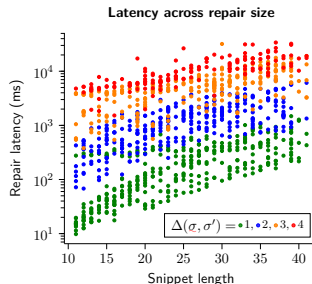
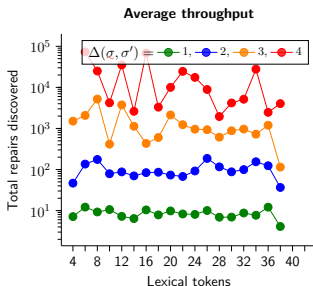
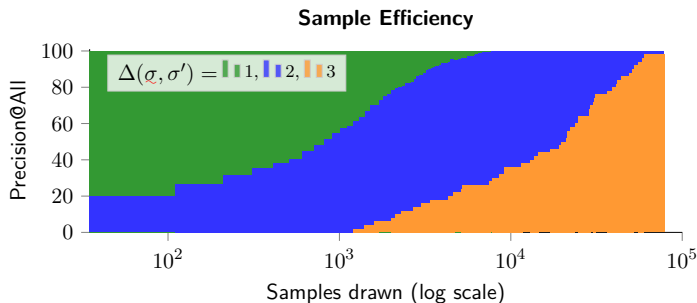


Figure: Latency benchmarks. Note the varying y-axis ranges. The red line marks Seq2Parse and the orange line marks BIFI's Precision@1 on the same repairs.

Results from sample efficiency experiments



Outcomes in the syntax repair pipeline

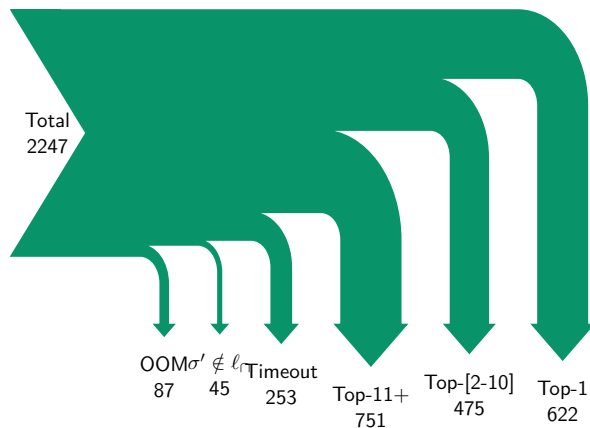


Figure: Sankey diagram of 967 total repair instances sampled uniformly from the StackOverflow Python dataset balanced across snippet lengths and edit distances ($(|\sigma|/10] \in [0, 8], \Delta(\sigma, \sigma') < 4)$) with a sampling timeout of 30s per repair.

Feature comparison matrix

	Sound	Complete	Natural	Theory		Tool
Tidyparse	✓	✓	✓	CFG _∩	✓	IDE-ready
Seq2Parse	✓ [†]	✗	✓	CFG	✗	Python
BIFI	✗	✗	✓	Σ*	✗	Python
OrdinalFix	✓	✗	✗	CFG+	✗	Rust
Outlines ¹	✓ [†]	✓ [†]	✓	EBNF	✗	Python
SynCode ¹	✓	✓	✓	EBNF	✗	Python
Aho/Irons	✓	✗	✗	CFG	✗	None

Sound = generated repairs always syntactically valid.

Complete = all valid repairs are eventually generated.

Natural \approx statistically likely / designed to model human preferability.

|| = Trivially parallelizable, i.e., designed to be executed on multiple cores.

¹ Not specifically intended for syntax repair, but can be adapted.

[†] Claimed by the authors, but counterexamples known to exist.

Abbreviated history of algebraic parsing

- Chomsky & Schützenberger (1959) - The algebraic theory of CFLs
- Cocke–Younger–Kasami (1961) - Bottom-up matrix-based parsing
- Brzozowski (1964) - Derivatives of regular expressions
- Valiant (1975) - first realizes the Boolean matrix correspondence
 - Naïvely, has complexity $\mathcal{O}(n^4)$, can be reduced to $\mathcal{O}(n^\omega)$, $\omega < 2.763$
- Lee (1997) - Fast CFG Parsing \iff Fast BMM, formalizes reduction
- Might et al. (2011) - Parsing with derivatives (Brzozowski \Rightarrow CFL)
- Bakinova, Okhotin et al. (2010) - Formal languages over GF(2)
- Bernady & Jansson (2015) - Certifies Valiant (1975) in Agda
- Cohen & Gildea (2016) - Generalizes Valiant (1975) to parse and recognize mildly context sensitive languages, e.g. LCFRS, TAG, CCG
- **Considine, Guo & Si (2022) - SAT + Valiant (1975) + holes**
- **Considine, Guo & Si (2024) - Levenshtein Bar-Hillel repairs**

Jin Guo, Xujie Si, David Bieber,
David Chiang, Brigitte Pientka, David Hui,
Ori Roth, Younesse Kaddar, Michael Schröder
Will Chrichton, Kristopher Micinski, Alex Lew
Matthijs Vákár, Michael Coblenz, Maddy Bowers



McGill
UNIVERSITY



Mila

Learn more at:

<https://tidyparse.github.io>