Syntax Repair as Language Intersection

ANONYMOUS AUTHOR(S)

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

1 INTRODUCTION

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

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

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

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

Thus we arrive at the core problem of this paper: how do we efficiently recover the most probable repairs in proximity to a syntactically broken code snippet? To address this problem, we propose to exhaustively evaluate every repair within a fixed edit distance and introduce an algorithm for doing so. Our algorithm constructs an expression denoting the language of nearby repairs, ranks its members by probability, then finally returns a small list of the most promising candidates.

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

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

2 BACKGROUND

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

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

Likewise, a finite state automaton (FSA) is a quintuple $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, and $I, F \subseteq Q$ are the set of initial and final states, respectively. We will adhere to this notation in the following sections.

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

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

$$e ::= \varnothing \mid \varepsilon \mid \Sigma \mid e \cdot e \mid e \vee e \mid e \wedge e$$

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

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

Brzozowski [1] introduces the concept of differentiation, which allows us to quotient a regular language by some given prefix.

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

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

Primarily, this gadget was designed to handle membership queries, for which purpose it has received considerable attention in recent years:

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

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

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

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

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

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

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

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

2.1 Language intersection

We will now define intersection in a slightly more expressive manner, which has the added benefit of being more readily parallelizable. Recall every regular language is context-free. Therefor, to take the intersection between two regular languages, we can treat one as a CFL, which often admits a more compact representation. Alternatively, we can take the intersection between a truly non-regular CFL (such as a programming language syntax) and some regular language.

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

Salomaa [5] introduces a direct, but inefficient construction:

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

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

however most synthetic productions in $P_\cap$ will be non-generating or unreachable. This naïve method will construct a synthetic production for state pairs which are not even connected by any path, which is clearly excessive.

3 INFORMAL STATEMENT

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

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

Given a lexical representation of a broken computer program $\underset{\sim}{\sigma}$ and a grammar G, our goal is to find every valid string σ consistent with the grammar G and within a certain edit distance, d. Consider the language of valid strings within a given edit distance of $\underset{\sim}{\sigma}$. We can intersect this langauge with the language of all valid programs, $\mathcal{L}(G)$. The resulting language ($\ell_\cap$) will contain every repair within the given edit distance. This language can be decoded in order of probability to retrieve the top-k most natural repairs. Finally, we map the sequence back to Unicode, add placeholders for fresh names, numbers, and string literals, then finally apply an off-the-shelf code formatter to display the repairs. Both the preprocessing and the formatting steps are tangential to this paper, in which we confine ourselves to a lexical alphabet.

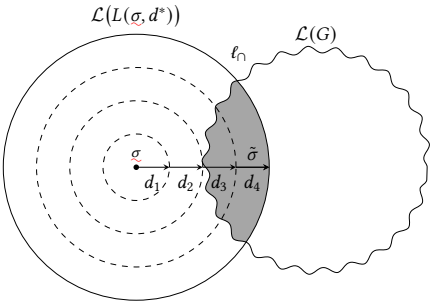


Fig. 1. CFL intersection with the local edit region of a given broken code snippet.

4 FORMAL STATEMENT

Definition 4.1 (Bounded Levenshtein-CFL reachability). Given a CFL, ℓ, and an invalid string, $\underset{\sim}{\sigma} : \bar{\ell}$, find every valid string reachable within d edits of $\underset{\sim}{\sigma}$, i.e., letting Δ be the Levenshtein metric and $L(\underset{\sim}{\sigma}, d) = \{\sigma' \mid \Delta(\underset{\sim}{\sigma}, \sigma') \leq d\}$ be the Levenshtein d-ball, we seek to find $\ell_\cap = L(\underset{\sim}{\sigma}, d) \cap \ell$.

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

Definition 4.2 (Ranked repair). Given a finite language $\ell_\cap = L(\underset{\sim}{\sigma}, d) \cap \ell$ and a probabilistic language model $\mathrm{P}_\theta : \Sigma^* \to [0, 1] \subset \mathbb{R}$, the ranked repair problem is to find the top-k maximum probability repairs under the language model. That is,

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

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

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

5 METHOD

Our method is to treat finite language intersections as matrix exponentiation.

Theorem 5.1. *For every CFG, G, and every acyclic NFA (ANFA), A, there exists a decision procedure* $\varphi : CFG \to ANFA \to \mathbb{B}$ *such that* $\varphi(G, A) \models [\mathcal{L}(G) \cap \mathcal{L}(A) \neq \varnothing]$ *which requires* $\mathcal{O}\big((\log |Q|)^c\big)$ *time using* $\mathcal{O}\big((|V||Q|)^k\big)$ *parallel processors for some* $c, k < \infty$.

Proof. WTS there exists a path $p \rightsquigarrow r$ in A such that $p \in I, r \in F$ where $p \rightsquigarrow r \vdash S$.

There are two cases, at least one of which must hold for $w \in V$ to parse a given $p \rightsquigarrow r$ pair:

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

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

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

$$M_0[r, c, w] = \bigvee_{a \in \Sigma} \{a \mid (w \to a) \in P \wedge (q_r \xrightarrow{a} q_c) \in \delta\} \tag{2}$$

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

$$[\ell \oplus r]_w = [\ell_w \vee r_w] \tag{3}$$

$$[\ell \otimes r]_w = \bigvee_{x,z \in V} \{\ell_x \cdot r_z \mid (w \to xz) \in P\} \tag{4}$$

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

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

To solve for the fixpoint , we can instead use exponentiation by squaring:

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

Therefor, we only need at most $\lceil \log_2 |Q| \rceil$ sequential steps to reach the fixpoint. Finally, we will union all the languages of every state pair deriving S into a new nonterminal, $S_\cap$.

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

To decode a witness in case of non-emptiness, one may simply `choose`$(S_\cap)$. □

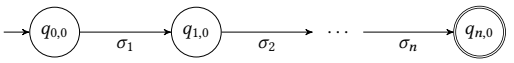
[2]Customarily, there is a $\frac{1}{k!}$ factor to suppress exploding entries, but alas this domain has no multiplicative inverse.

6 EXAMPLES

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

6.1 Recognition as intersection

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



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

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

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

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

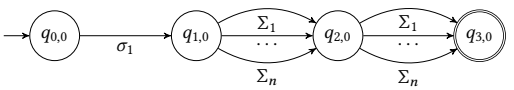
Once the fixpoint M_∞ is attained, the proposition $[S \in \Lambda^*_\sigma]$ decides language membership, i.e., $[\sigma \in \mathcal{L}(G)]$ [3]. So far, this procedure is essentially the textbook CYK algorithm in a linear algebraic notation [3] and a well-established technique in the parsing literature [4].

6.2 Completion as intersection

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

Definition 6.1 (Completion). Let $\underline{\Sigma} = \Sigma \cup \{\_\}$, where _ denotes a hole. We denote $\sqsubseteq : \Sigma^n \times \underline{\Sigma}^n$ as the relation $\{\langle \sigma', \sigma \rangle \mid \sigma_i \in \Sigma \implies \sigma'_i = \sigma_i\}$ and the set of all inhabitants $\{\sigma' : \Sigma^+ \mid \sigma' \sqsubseteq \sigma\}$ as $\mathrm{H}(\sigma)$. Given a *porous string*, $\sigma : \underline{\Sigma}^*$ we seek all syntactically valid inhabitants, i.e., $A(\sigma) = \mathrm{H}(\sigma) \cap \ell$.

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



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

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

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

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

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

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

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

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

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

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

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

6.3 Repair as intersection

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

As the original construction defined by Schultz and Mihov [6] contains cycles and ε-transitions, we propose a variant which is ε-free and acyclic. Furthermore, we adopt a nominal form which supports infinite alphabets and considerably simplifies the language intersection to follow. Illustrated in Fig. 2 is an example of a small Levenshtein automaton recognizing $L(\sigma : \Sigma^5, 3)$. Unlabeled arcs accept any terminal from the alphabet, Σ. Equivalently, this transition system can be viewed as a kind of proof system within an unlabeled lattice. The following construction is equivalent to Schultz and Mihov's original Levenshtein automaton, but is more amenable to our purposes as it does not any contain ε-arcs, and instead uses skip connections to recognize consecutive deletions of varying lengths.

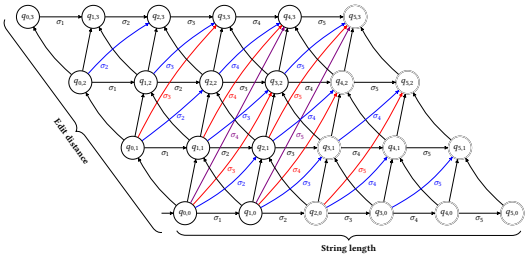


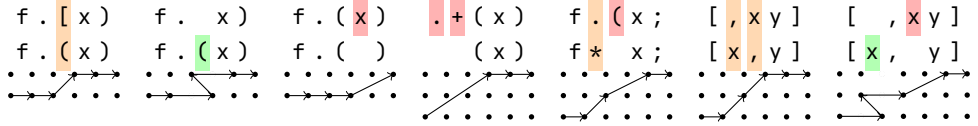
Fig. 2. NFA recognizing Levenshtein $L(\sigma : \Sigma^5, 3)$.

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

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

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

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



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

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

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

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

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

As a concrete example, suppose we have the string, $\sigma =$ ( )) and wish to balance the parentheses. We will initially have the Levenshtein automaton, A, depicted in Fig. 3. To check for non-emptiness, we will perform the following procedure. Suppose we have a CNF CFG, $G' = \{S \to LR, S \to LF, S \to SS, F \to SR, L \to (, R \to)\}$ and let us assume an ordering of S, F, L, R on V.

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

Under such an ordering, the adjacency matrix takes an upper triangular form and becomes the template for the initial parse chart, M_0 (Fig. 6). Each entry of this chart corresponds to a vector of expressions $E^{|V|}$ with at least one expression denoting a nonempty language. Likewise, the reachability matrix signifies a subset of state pairs which can participate in the language intersection. The adjacency and reachability matrices will always cover the expression vectors of the initial and final parse charts, respectively. In other words, we may safely ignore absent $\langle q, q' \rangle$ pairs in the reachability matrix, as these state pairs definitely cannot participate in the intersection.

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

Just as before, we will define $\boxplus, \boxtimes$ over GRE vectors, where $X \boxtimes Z = [X_x \cdot Z_z \mid (w \to xz) \in P]_{w \in V}$ and $X \boxplus Z = [X_x \vee Z_z]_{w \in V}$. Finally, we will repeat the matrix exponential, using M_∞ in the binary domain as a guide. This allows us to construct the regular expression tree for $G_\cap = q_{00} S q_{20} \vee q_{00} S q_{31}$ shown in Fig. 8. Once this regex is constructed, decoding becomes simply a matter of invoking `choose`$(G_\cap)$ to produce a concrete repair.

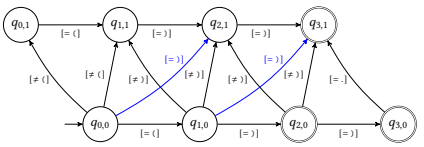


Fig. 3. Simple Levenshtein automaton.

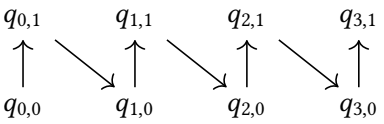


Fig. 4. Pairing function over $L(\sigma : \Sigma^3, 1)$.

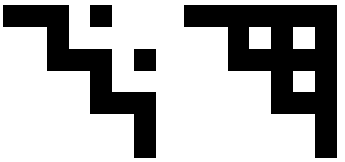


Fig. 5. Adjacency and reachability matrix.

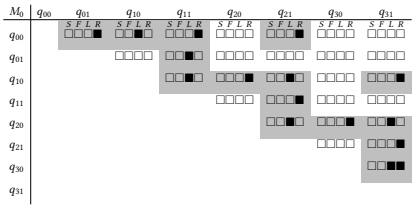


Fig. 6. Initial parse chart configuration.

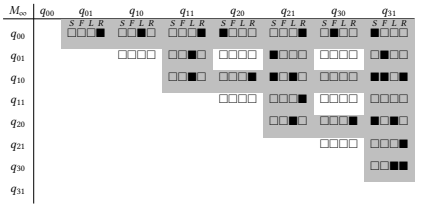


Fig. 7. Final parse chart configuration.

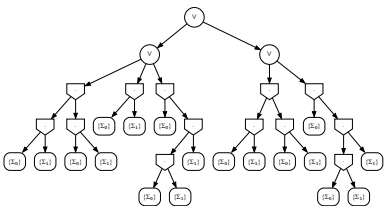


Fig. 8. Regular expression denoting $\mathcal{L}(G_\cap)$.

7 MEASURING THE LANGUAGE INTERSECTION

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

7.1 Exact enumeration

A brute force solution would be to generate every path and rank every one by its probability. It should be obvious why is unviable due its worst case complexity, but bears mentioning due to its global optimality. In certain cases, it can be realized when the intersection language is small.

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

Definition 7.1 (Cardinality). $$|e| : E \to \mathbb{N} = \begin{cases} 1 & \text{if } R \in \Sigma \\ x \times z & \text{if } e = x \cdot z \\ x + z & \text{if } e = x \vee z \end{cases}$$

Theorem 7.2 (Enumeration). *To enumerate, invoke* $\bigcup_{i=0}^{|R|} \{\mathit{enum}(R, i)\}$*:*

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

7.2 Mode collapse

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

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

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

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

7.3 Ambiguity

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

Lemma 7.3. *If the FSA, α, is ambiguous, then the intersection grammar, $G_\cap$, can be ambiguous.*

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

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

8 IMPLEMENTATION

The implementation consists of five stages. We will make the simplifying assumption that each GPU kernel is a pure function that takes as input a coordinate triple $r, c, v : \mathbb{N}$ and one or more flat buffers $b_1 : \mathbb{N}^{d_1}, \ldots b_n : \mathbb{N}^{d_n}$, does some arithmetic, and returns a single buffer $b_{\text{out}} : \mathbb{N}^d$.

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

(1) `lev_build` : $\Sigma^{|Q|-1} \times \mathbb{N}^3 \to$ NFA – constructs a Levenshtein NFA from the broken string.
(2) `cfl_fixpt` : NFA $\times$ CFG $\to \mathbb{B}^{|Q|\times|Q|\times|V|}$ – computes the matrix exponential.
(3) `reg_build` : $\mathbb{B}^{|Q|\times|Q|\times|V|} \times$ CFG $\to$ REG – constructs the regular expression for $G_\cap$.
(4) `reg_dcode` : REG $\to (\Sigma^* \times \mathbb{N})^{p \gg 1}$ – generates and scores a large number of repairs.
(5) `sel_top_k` : $(\Sigma^* \times \mathbb{N})^{p \gg 1} \to (\Sigma^*)^{k \ll p}$ – returns a small set of the most probable repairs.

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

The REG datatype is slightly more complex to flatten, as being an algebraic datatype it can be simplified in various ways....

We will define the serial version of the code for each procedure below:

Algorithm 1 `lev_build` pseudocode

Require: $T : \mathbb{T}_2$ intersection grammar, PCFG Score: $\mathbb{T} \to \mathbb{R}$
1: $\hat{A} \leftarrow \varnothing$, `seed` $\leftarrow 0$ ▷ Initialize set of parse trees.
2: **for** `seed` $< |T|$ and uninterrupted **do**
3: $t \leftarrow \varphi'(T, \texttt{seed++})$ ▷ Draw unique $\mathbb{Z}_{|T|}$ and decode into fresh parse tree.
4: $\hat{A} \leftarrow \hat{A} \cup \{t\}$
5: **return** $[\mathscr{D}(a) \mid a \in \hat{A}$ **ranked by** Score$(a)]$ ▷ Rerank by PCFG likelihood and defoliate.

Algorithm 2 `cfl_fixpt` pseudocode

Require: $T : \mathbb{T}_2$ intersection grammar, PCFG Score: $\mathbb{T} \to \mathbb{R}$
1: $\hat{A} \leftarrow \varnothing$, `seed` $\leftarrow 0$ ▷ Initialize set of parse trees.
2: **for** `seed` $< |T|$ and uninterrupted **do**
3: $t \leftarrow \varphi'(T, \texttt{seed++})$ ▷ Draw unique $\mathbb{Z}_{|T|}$ and decode into fresh parse tree.
4: $\hat{A} \leftarrow \hat{A} \cup \{t\}$
5: **return** $[\mathscr{D}(a) \mid a \in \hat{A}$ **ranked by** Score$(a)]$ ▷ Rerank by PCFG likelihood and defoliate.

Algorithm 3 `reg_build` pseudocode

Require: $T : \mathbb{T}_2$ intersection grammar, PCFG Score: $\mathbb{T} \to \mathbb{R}$

$\hat{A} \leftarrow \varnothing$, `seed` $\leftarrow 0$ ▷ Initialize set of parse trees.
for `seed` $< |T|$ and uninterrupted **do**
  $t \leftarrow \varphi'(T, \texttt{seed}\texttt{++})$ ▷ Draw unique $\mathbb{Z}_{|T|}$ and decode into fresh parse tree.
  $\hat{A} \leftarrow \hat{A} \cup \{t\}$
return $[\mathbb{P}(a) \mid a \in \hat{A}$ **ranked by** Score$(a)]$ ▷ Rerank by PCFG likelihood and defoliate.

Algorithm 4 `reg_dcode` pseudocode

Require: $T : \mathbb{T}_2$ intersection grammar, PCFG Score: $\mathbb{T} \to \mathbb{R}$

$\hat{A} \leftarrow \varnothing$, `seed` $\leftarrow 0$ ▷ Initialize set of parse trees.
for `seed` $< |T|$ and uninterrupted **do**
  $t \leftarrow \varphi'(T, \texttt{seed}\texttt{++})$ ▷ Draw unique $\mathbb{Z}_{|T|}$ and decode into fresh parse tree.
  $\hat{A} \leftarrow \hat{A} \cup \{t\}$
return $[\mathbb{P}(a) \mid a \in \hat{A}$ **ranked by** Score$(a)]$ ▷ Rerank by PCFG likelihood and defoliate.

Algorithm 5 `sel_top_k` pseudocode

Require: $T : \mathbb{T}_2$ intersection grammar, PCFG Score: $\mathbb{T} \to \mathbb{R}$

$\hat{A} \leftarrow \varnothing$, `seed` $\leftarrow 0$ ▷ Initialize set of parse trees.
for `seed` $< |T|$ and uninterrupted **do**
  $t \leftarrow \varphi'(T, \texttt{seed}\texttt{++})$ ▷ Draw unique $\mathbb{Z}_{|T|}$ and decode into fresh parse tree.
  $\hat{A} \leftarrow \hat{A} \cup \{t\}$
return $[\mathbb{P}(a) \mid a \in \hat{A}$ **ranked by** Score$(a)]$ ▷ Rerank by PCFG likelihood and defoliate.

REFERENCES

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

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

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

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

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

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

A LEVENSHTEIN AUTOMATA MATRICES

These are useful for visually checking different implementations.

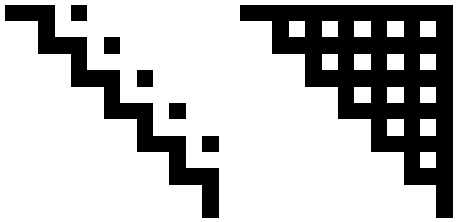


Fig. 9. Lev($|\sigma|$=6, Δ=1) adjacency and reachability matrices.

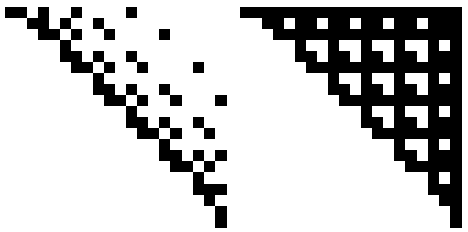


Fig. 10. Lev($|\sigma|$=6, Δ=2) adjacency and reachability matrices.

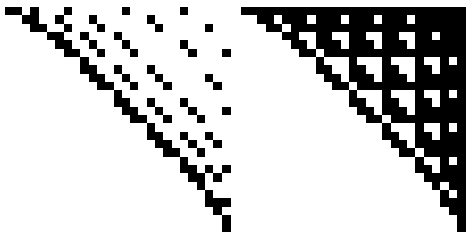


Fig. 11. Lev($|\sigma|$=6, Δ=3) adjacency and reachability matrices.

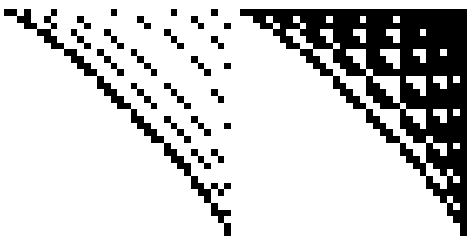


Fig. 12. Lev($|\sigma|$=6, Δ=4) adjacency and reachability matrices.

B LEVENSHTEIN AUTOMATA MINIMALITY

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

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

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

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

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

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

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

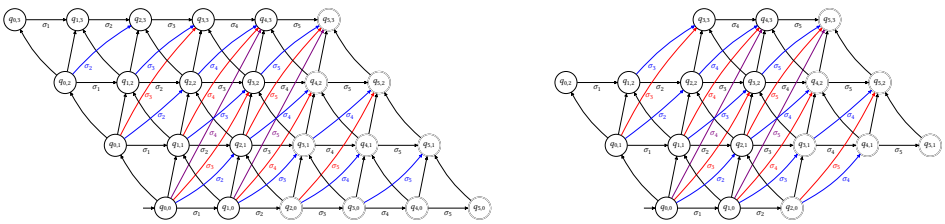


Fig. 13. Levenshtein NFA before and after pruning.

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

`_ _ + ) ]` ❌ ∧ `_ _ _ ) ]` ✅

`[ ( + _ _` ❌ ∧ `[ ( _ _ _` ✅