Tidyparse: Real-Time Context Free Error Correction

Breandan Mark Considine
McGill University
bre@ndan.co

Jin Guo
McGill University
jguo@cs.mcgill.ca

Xujie Si
McGill University
xsi@cs.mcgill.ca

Abstract

Tidyparse is a program synthesizer that performs real-time error correction for context free languages. Given both an arbitrary context free grammar (CFG) and an invalid string, the tool lazily generates admissible repairs while the author is typing, ranked by Levenshtein edit distance. Repairs are guaranteed to be sound, complete, syntactically valid and minimal. Tidyparse is the first system of its kind offering these guarantees in a real-time editor. To accelerate code completion, we design and implement a novel incremental parser-synthesizer that transforms CFGs onto a dynamical system over finite field arithmetic, enabling us to suggest syntax repairs in-between keystrokes. We have released an IDE plugin demonsrating the system described.[1]

1 Introduction

Modern research on error correction can be traced back to the early days of coding theory, when researchers designed *error-correcting codes* (ECCs) to denoise transmission errors induced by external interference, whether due to collision with a high-energy proton, manipulation by an adversary or some typographical mistake. In this context, *code* can be any logical representation for communicating information between two parties (such as a human and a computer), and an ECC is a carefully-designed code which ensures that even if some portion of the message should be corrupted through accidental or intentional means, one can still recover the original message by solving a linear system of equations. In particular, we frame our work inside the context of errors arising from human factors in computer programming.

In programming, most such errors initially manifest as syntax errors, and though often cosmetic, manual repair can present a significant challenge for novice programmers. The ECC problem may be refined by introducing a language, $\mathcal{L} \subset \Sigma^*$ and considering admissible edits transforming an arbitrary string, $s \in \Sigma^*$ into a string, $s' \in \mathcal{L}$. Known as *error-correcting parsing* (ECP), this problem was well-studied in the early parsing literature, cf. Aho and Peterson [1], but fell out of favor for many years, perhaps due to its perceived complexity. By considering only minimal-length edits, ECP can be reduced to the so-called *language edit distance* (LED) problem, recently shown to be subcubic [2], suggesting its possible tractability. Previous results on ECP and LED were primarily of a theorietical nature, but now, thanks to our contributions, we have finally realized a practical prototype.

[1]https://plugins.jetbrains.com/plugin/19570-tidyparse

2 Toy Example

Suppose we are given the following context free grammar:

```
S -> S and S | S or S | ( S ) | true | false | ! S
```

For reasons that will become clear in the following section, this is automatically rewritten into the equivalent grammar:

```
 F.! → !   ε+ → ε       S → false    F.and → and
 F.( → (   ε+ → ε+ ε+   S → F.! S      S.) → S F.)
 F.) → )    S → <S>     S → S or.S    or.S → F.or S
 F.ε → ε    S → true    S → S and.S  and.S → F.and S
F.or → or   S → S ε+    S → F.( S.)
```

If provided with a string containing a number of holes, our tool will return a dozen completions in a few milliseconds:

```
true _ _ _ ( false _ ( _ _ _ _ ! _ _ ) _ _ _ _
```

```
true or ! ( false or ( <S> ) or ! <S> ) or <S>
true or ! ( false and ( <S> ) or ! <S> ) or <S>
true or ! ( false and ( <S> ) and ! <S> ) or <S>
true or ! ( false and ( <S> ) and ! <S> ) and <S>
true and ( false and ( <S> ) and ! ! <S> ) and <S>
true or ! ( false or ( ! <S> ) or ! <S> ) or <S>
true or ! ( false or ( <S> ) or ! ! <S> ) and <S>
true or ! ( false and ( <S> ) or ! ! <S> ) and <S>
true or ! ( false and ( ! <S> ) and ! <S> ) and <S>
true and ! ( false and ( <S> ) and ! ! <S> ) and <S>
true or ! ( false or ( ! <S> and ! ! <S> ) or <S> )
true or ( ( false and ( <S> ) and ! ! <S> ) or <S> )
```

Similarly, if provided with a string containing various errors, Tidyparse will return several suggestions how to fix it, where green is insertion, orange is substitution and red is deletion.

```
true and ( false or and true false
```

```
1.) true and ( false or ! true )
2.) true and ( false or <S> and true )
3.) true and ( false or ( true ) )
4.) true and ( false or ! ! false )
5.) true and ( false or <S> or false )
6.) true and ( false or ! <S> and true )
7.) true and ( false or ! ( false ) )
8.) true and ( false or ! true ) or <S>
9.) true and ( false or ! <S> ) and true false
```

In the following paper, we will describe how we built it.

3 Matrix Theory

We recall that a CFG is a quadruple consisting of terminals, Σ, nonterminals, V, productions, $P : V \to (V \mid \Sigma)^*$, and the start symbol, S. It is a well-known fact that every CFG can be reduced to *Chomsky Normal Form* (CNF), $P^* : V \to (V^2 \mid \Sigma)$, in which every production takes one of two forms, either $v_0 \to v_1 v_2$, or $v_0 \to \sigma$, where $v_{0,1,2} : V$ and $\sigma : \Sigma$. For example, we can rewrite the CFG $\{S \to SS \mid (S) \mid ()\}$, into CNF as:

$$\{S \to XR \mid SS \mid LR,\ L \to (,\ R \to),\ X \to LS\}$$

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

$$a \otimes b := \{C \mid \langle A, B\rangle \in a \times b, (C \to AB) \in P\} \qquad (1)$$

We initialize $\mathbf{M}^0_{r,c}(\mathcal{G}, \sigma) := \{V \mid c = r + 1, (V \to \sigma_r) \in P\}$ and search for a matrix $\mathbf{M}^*$ via fixpoint iteration,

$$\mathbf{M}^* = \begin{pmatrix} \varnothing & \{V\}_{\sigma_1} & \cdots & \mathcal{T} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \{V\}_{\sigma_n} \\ \varnothing & \cdots & \cdots & \varnothing \end{pmatrix} \qquad (2)$$

where $\mathbf{M}^*$ is the least solution to $\mathbf{M} = \mathbf{M} + \mathbf{M}^2$. We can then define the recognizer as: $S \in \mathcal{T}? \iff \sigma \in \mathcal{L}(\mathcal{G})?$

While theoretically elegant, this decision procedure can be optimized by lowering onto a rank-3 binary tensor. We do so simply by noting that $\bigoplus_{k=1}^{n} \mathbf{M}_{ik} \otimes \mathbf{M}_{kj}$ has cardinality bounded by $|V|$ and is thus representable as a fixed-length vector using the characteristic function, $\mathbb{1}$. In particular, $\oplus, \otimes$ are defined as $\boxplus, \boxtimes$, so that the following diagram commutes:

$$\begin{tikzcd}
V \times V \arrow[r, "\oplus/\otimes"] \arrow[d, shift left, "\mathbb{1}^2"] & V \arrow[d, shift left, "\mathbb{1}"] \\
\mathbb{B}^{|V|} \times \mathbb{B}^{|V|} \arrow[r, "\boxplus/\boxtimes"'] \arrow[u, shift left, "\mathbb{1}^{-2}"] & \mathbb{B}^{|V|} \arrow[u, shift left, "\mathbb{1}^{-1}"]
\end{tikzcd}$$

The compactness of this representation can be improved via a combinatorial number system without loss of generality, although $\mathbb{1}$ is a convenient encoding for SAT. By allowing the matrix $\mathbf{M}^0_{r,c}$ to contain bitvector variables $\mathcal{B}^{|V|}$ representing holes in the string and associated nonterminals, we obtain a set of multilinear SAT equations whose solutions exactly correspond to the set of admissible repairs and their corresponding parse forests. This is described further in §4.

Full details of the bisimilarity between parsing and matrix multiplication can be found in Valiant [4], who shows its time complexity to be $\mathcal{O}(n^\omega)$ where ω is the matrix multiplication bound, and Lee [3], showing that speedups to Boolean matrix multiplication may be translated back into CFG parsing. By assuming sparsity, this technique can typically be reduced to linearithmic time, and is believed to be the most efficient procedure for CFL recognition to date.

3.1 Sampling k-combinations without replacement

Let $\mathbf{M} : \mathrm{GF}(2^{n\times n})$ be a matrix with the structure $\mathbf{M}^0_{r,c} = P_c$ if $r = 0$ else $\mathbb{1}[c = r - 1]$, where P is a feedback polynomial over $GF(2^n)$ with coefficients $P_{1\ldots n}$ and semiring operators $\oplus := \veebar, \otimes := \wedge$:

$$\mathbf{M}^t V = \begin{pmatrix} P_1 & \cdots & \cdots & \cdots & P_n \\ \top & \circ & \cdots & \cdots & \circ \\ \circ & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \circ & \cdots & \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} V_1 \\ \vdots \\ \vdots \\ \vdots \\ V_n \end{pmatrix} \qquad (3)$$

Selecting any $V \neq \mathbf{0}$ and coefficients $P_{1\ldots n}$ from a known *primitive polynomial* then powering the matrix $\mathbf{M}$ generates an ergodic sequence over $\mathrm{GF}(2^n)$:

$$\mathbf{S} = \begin{pmatrix} V & \mathbf{M}V & \mathbf{M}^2 V & \mathbf{M}^3 V & \cdots & \mathbf{M}^{2^n-1} V \end{pmatrix} \qquad (4)$$

This sequence has *full periodicity*, in other words, for all $i, j \in [0, 2^n), \mathbf{S}_i = \mathbf{S}_j \Rightarrow i = j$.

To uniformly sample $\boldsymbol{\sigma} \sim \Sigma^n$ without replacement, we could track historical samples and do rejection sampling, or, we can form an injection $GF(2^n) \rightharpoonup \Sigma^d$, cycle a primitive polynomial over $GF(2^n)$, then discard samples that do not identify an element in any indexed dimension. This procedure rejects $(1 - |\Sigma| 2^{-\lceil \log_2 |\Sigma| \rceil})^d$ samples on average and requires $\sim \mathcal{O}(1)$ per sample and $\mathcal{O}(2^n)$ to exhaustively search the space.

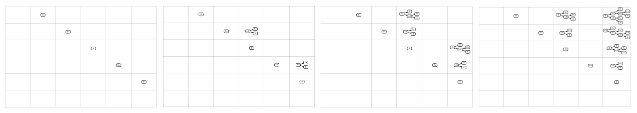
For example, if we wanted to sample $\Sigma^2 = \{A, B, C\}^2$, we can use the primitive polynomial $x^4 + x^3 + 1$

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
1000	0100	0010	1001	1100	0110	1011	0101
C A	B A	A C	C B		B C		B B

We will use this technique to lazily sample k-combinations without replacement as described in §5.

4 SAT Encoding

Specifically, the repairs occur along holes in the superdiagonal $\mathbf{M}^*_{r+1=c}$, and the upper-triangular entries $\mathbf{M}^*_{r+1<c}$ represent the corresponding parse forests. If no solution exists, then the upper triangular entries will appear as a jagged-shaped ridge whose peaks represent the roots of the parsable subtree. We illustrate this fact in §5:



We precompute the shadow of fully-resolved substrings before feeding it to the SAT solver. If the substring is known, we can simply compute this directly outside the SAT solver. Shadow regions are bitvector literals and light regions correspond to bitvector variables.

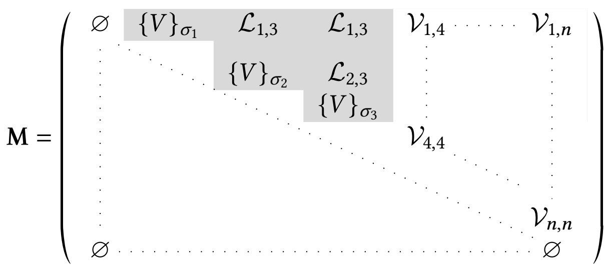


Figure 1. SAT tensor representing the string $\sigma_1\ \sigma_2\ \sigma_3$ _ … _ in which shaded regions demarcate known bitvector literals $\mathcal{L}_{r,c}$ (i.e., representing established nonterminal forests) and unshaded regions correspond to bitvector variables $\mathcal{V}_{r,c}$ (i.e., representing seeded nonterminal forests to be grown). Since $\mathcal{L}_{r,c}$ are fixed, we precompute them outside the SAT solver.

5 Error Recovery

Unlike classical parsers which totally fail on an error or need special support for error-recovery, if the tree cannot parse, we can do error recovery with partial subtrees. These subtrees provide a natural debugging environment to aid the repair process.

```
true and true ! and false or true ! true and false
```

```
Parseable subtrees (3 leaves / 2 branches)

 🌿            🌿               🌿
 └── ! [3]     └── and [4]     └── or [6]

 🌿                            🌿
 └── S [0..2]                  └── S [8..11]
     ├── true [0]                  ├── S [8..9]
     ├── and [1]                   │   ├── ! [8]
     └── true [2]                  │   └── true [9]
                                   ├── and [10]
                                   └── false [11]
```

These branches correspond to peaks on the upper triangular (UT) matrix ridge. We traverse the peaks by elevation from highest to lowest to collect the partial AST branches.

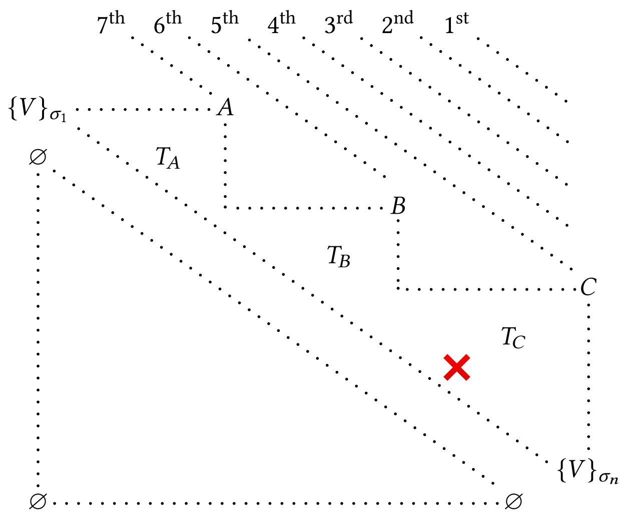


Figure 2. Peaks along the UT matrix ridge correspond to maximally parseable substrings. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover the partial AST branches. The example depicted above contains three such branches, rooted at nonterminals C, B, A.

6 Realtime Error Correction

So we have a procedure $P : \mathcal{G} \times \Sigma^d \rightarrow \{\Sigma^d\}$. But where do we put the holes? For a given number of holes, k, there are roughly $\binom{n}{k}$ possible repairs. In practice the cardinality of this space can be very large. In order to provide suggestions in realtime, we generate the repairs according to the following procedure:

1. Fetch the most recent grammar and string from the editor.
2. We exclude parsable substrings from hollowing.
3. Lazily enumerate all possible hole configurations of increasing length.
4. For each size, sample uniformly without replacement using a Galois field.
5. Hole configurations are prioritized by distance to caret location.
6. Hole configurations are prioritized by distance to fishy locations.
7. Hole configurations are translated to sketch templates using the procedure described in §6.1.
8. We feed the sketch templates to the incremental SAT solver.
9. The first dozen results are decoded and displayed to the user in order of increasing Levenshtein distance.

The entire procedure is lazy and intermediate results are cached to avoid recomputation. Incoming keystrokes interrupt the solver and reset the process back to step 1.

6.1 Deletion, Substitution, Insertion

Deletion, substitution and insertion can be simulated by first adding a left- and right- ε-production to each unit production:

$$\frac{\Gamma \vdash \varepsilon \in \Sigma}{\Gamma \vdash (\varepsilon^+ \rightarrow \varepsilon \mid \varepsilon^+ \, \varepsilon^+) \in P}\ \varepsilon\text{-DUP}$$

$$\frac{\Gamma \vdash (A \rightarrow B) \in P}{\Gamma \vdash (A \rightarrow B\,\varepsilon^+ \mid \varepsilon^+\,B \mid B) \in P}\ \varepsilon^+\text{-INT}$$

To generate the sketch templates, we substitute two holes at each index to be replaced, $H(\sigma, i) = \sigma_{1\ldots i-1}\ \_\ \_\ \sigma_{i+1\ldots n}$, letting $\sigma' \leftarrow H(\sigma, i)$ and invoke $P(\mathcal{G}, H(\sigma, i))$. Five outcomes are then possible:

$$\sigma_1 \ldots \sigma_{i-1}\ \gamma_1\gamma_2\ \sigma_{i+1} \ldots \sigma_n, \gamma_{1,2} = \varepsilon \tag{5}$$

$$\sigma_1 \ldots \sigma_{i-1}\ \gamma_1\gamma_2\ \sigma_{i+1} \ldots \sigma_n, \gamma_1 \neq \sigma_i, \gamma_2 = \varepsilon \tag{6}$$

$$\sigma_1 \ldots \sigma_{i-1}\ \gamma_1\gamma_2\ \sigma_{i+1} \ldots \sigma_n, \gamma_1 = \varepsilon, \gamma_2 \neq \sigma_i \tag{7}$$

$$\sigma_1 \ldots \sigma_{i-1}\ \gamma_1\gamma_2\ \sigma_{i+1} \ldots \sigma_n, \gamma_1 = \sigma_i, \gamma_2 \neq \varepsilon \tag{8}$$

$$\sigma_1 \ldots \sigma_{i-1}\ \gamma_1\gamma_2\ \sigma_{i+1} \ldots \sigma_n, \gamma_1 \notin \{\varepsilon, \sigma_i\}, \gamma_2 = \sigma_i \tag{9}$$

Eq. (5) corresponds to deletion, eqs. (6, 7) correspond to substitution, and eqs. (8, 9) correspond to insertion. The procedure is repeated for all indices in the replacement set. The solutions returned by the solver will be strictly equivalent to handling each edit operation separately.

7 Implementation

Tidyparse accepts a CFG and a string to parse and returns a set of candidate strings, ordered by their Levenshtein edit distance to the original string. Our method lowers the CFG and candidate string onto a matrix dynamical system using an extended version of Valiant's construction and solves for the fixpoint matrix using an incremental SAT solver.

Here is how we implemented it (and so can you) [4].

8 Examples

There are some more examples too.

The line between parsing and computation is blurry.

9 Conclusion

Our approach to parsing has many advantages.

- Error correction.
- Program repair.
- Program synthesis.
- Parsing with holes.
- Naturally integrates with masked language model (MLM)-based neural program repair.
- Parsing with natural error recovery.
- Helps to facilitate language learning.
- GPU acceleration.

10 Acknowledgements

The first author would like to thank his co-advisor Xujie Si for providing many helpful suggestions during the development of this project, including the optimized fixpoint, test cases, and tree denomalization procedure, and collaborator Nghi Bui for early feedback on the plugin.

References

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

[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] Lillian Lee. 2002. Fast context-free grammar parsing requires fast boolean matrix multiplication. Journal of the ACM (JACM) 49, 1 (2002), 1–15. https://arxiv.org/pdf/cs/0112018.pdf

[4] Leslie G Valiant. 1975. General context-free recognition in less than cubic time. Journal of computer and system sciences 10, 2 (1975), 308–315. http://people.csail.mit.edu/virgi/6.s078/papers/valiant.pdf