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.

2 Toy Example

Suppose we are given the following context free grammar:

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

This gets rewritten into the following CNF 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.)
```

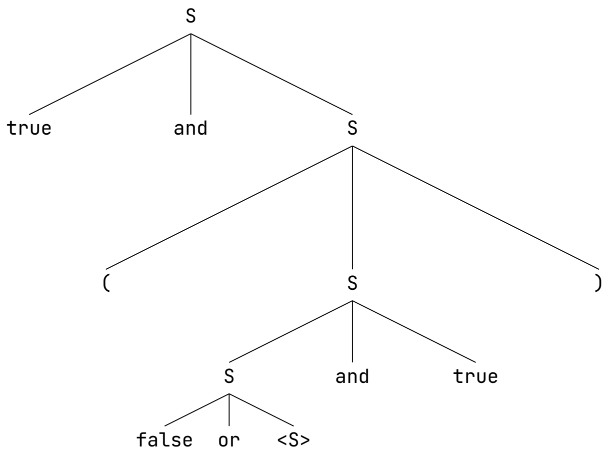
This CFG corresponds to the Dyck-3 language of balanced parentheses. Suppose we have the following string:

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

Our tool produces the following suggestions, where green is insertion, orange is substitution and red is deletion.

```
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
```

Selecting the third entry will produce a single parse tree:



All the above visualizations were generated automatically.

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

3 Matrix Theory

We recall that a CFG is a quadruple consisting of terminals, Σ, nonterminals, V, productions, $P : V \rightarrow (V \mid \Sigma)^*$, and a start symbol, S. It is a well-known fact that every CFG can be reduced to *Chomsky Normal Form* (CNF), $P^* : V \rightarrow (V^2 \mid \Sigma)$, in which every production takes one of two forms, either $v_0 \rightarrow v_1 v_2$, or $v_0 \rightarrow \sigma$, where $v_{0\dots2} : V$ and $\sigma : \Sigma$. For example, we can rewrite the CFG $\{S \rightarrow SS \mid (S) \mid ()\}$, into CNF as:

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

Given a CFG, $\mathcal{G} : \Sigma, \langle V, P, S\rangle$ in CNF, we can construct a recognizer $R_{\mathcal{G}} : \Sigma^n \rightarrow \mathbb{B}$ 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:

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

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

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

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:

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

We choose a convenient encoding, however it can be improved using a combinatorial number system.

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 currently the best known asymptotic bound for CFL recognition to date.

By allowing the matrix $\mathbf{M}^0_{r,c}$ to contain bitvector variables $\mathcal{B}^{|V|}$ representing holes in the string, we obtain a set of multilinear equations whose solutions exactly correspond to the set of admissible repairs and their corresponding parse trees. This is described in further detail in §4.

3.1 Sampling k-combinations without replacement

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

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

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

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

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

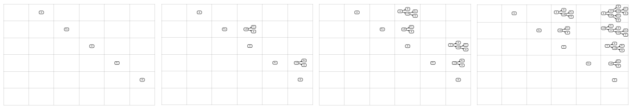
To uniformly sample $\boldsymbol{\sigma} \sim \Sigma^n$ without replacement, we could track historical samples and do rejection sampling, or, we can form an injection $GF(2^n) \rightharpoonup \Sigma^d$, cycle a primitive polynomial over $GF(2^n)$, then discard samples that do not identify an element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

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

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 tree. 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 the following example:



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. TODO: describe procedure.

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. The model can explain which trees parse and which do not.

```
if ( true and false ) then if true then 3
```

```
Parseable subtrees (4 leaves / 2 branches):

  🌿                        🌿
  └── if [0]                └── then [6]

  🌿                        🌿
  └── else [13]             └── false [14]

  🌿                        🌿
  └── B [1..5]              └── I [7..12]
      ├── ( [1]                 ├── if [7]
      ├── B [2..4]              ├── true [8]
      │   ├── true [2]          ├── then [9]
      │   ├── and [3]           ├── 3 [10]
      │   └── false [4]         ├── else [11]
      └── ) [5]                 └── 2 [12]
```

These correspond to peaks on the UT matrix ridge. We traverse the peaks according to their orthogonal elevation from the diagonal, from highest to lowest.

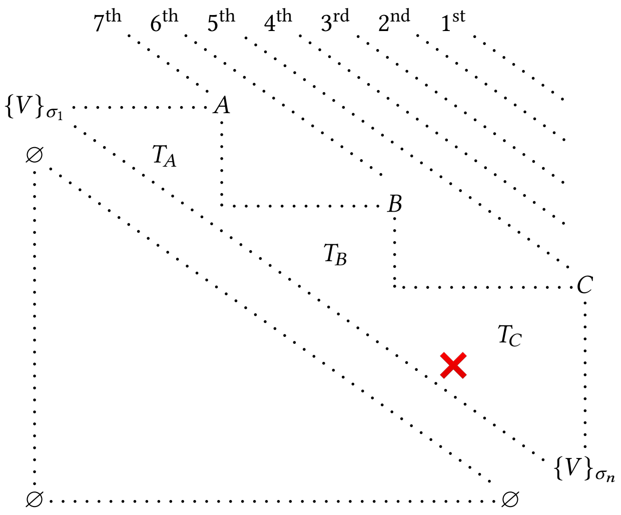


Figure 1. Mountains in the ridge structure correspond to parseable substrings. We iterate over upper diagonals in decreasing altitude and discard any subtree that falls totally within the shadow of another. The example shown above has three branches, rooted at nonterminals C, B, A.

6 Hollowing Procedure

So we have a procedure $P : \mathcal{G} \times \Sigma^d \longrightarrow \{\Sigma^d\}$. 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 sample without replacement from this space, we generate the repairs in the following order.

1. We remove parsable substrings from hollowing.
2. Lazily enumerate all possible hole configurations of increasing length.
3. For each size, sample uniformly without replacement using a Galois field.
4. Hole configurations are prioritized by distance to caret location.
5. Hole configurations are prioritized by distance to fishy locations.
6. We feed the hole configurations to the incremental SAT solver.
7. 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.

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...

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.

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