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 complete, grammatically consistent 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 demonstrating 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 theoretical 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.)
```

Given a string containing holes such as the one below, Tidyparse will return several 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>
...
```

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 ) )
...
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, $\mathcal{G}' : \langle \Sigma, V, P, S \rangle$ 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:

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

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,

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

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

Note 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:

$$\begin{array}{ccc} V \times V & \xrightarrow{\oplus/\otimes} & V \\ \uparrow{\scriptstyle \mathbb{1}^{-2}}\ \downarrow{\scriptstyle \mathbb{1}^{2}} & & \uparrow{\scriptstyle \mathbb{1}^{-1}}\ \downarrow{\scriptstyle \mathbb{1}} \\ \mathbb{B}^{|V|} \times \mathbb{B}^{|V|} & \xrightarrow[\boxplus/\boxtimes]{} & \mathbb{B}^{|V|} \end{array}$$

Full details of the bisimilarity between parsing and matrix multiplication can be found in Valiant [5], who shows its time complexity to be $\mathcal{O}(n^\omega)$ where ω is the matrix multiplication bound ($\omega < 2.77$), and Lee [4], who shows that speedups to Boolean matrix multiplication are realizable by CFL parsers.

3.1 Sampling k-combinations without replacement

Let $\mathbf{M} : \mathrm{GF}(2^{n \times n})$ be a matrix whose structure is depicted in Eq. 3, where P is a feedback polynomial over $GF(2^n)$ with coefficients $P_{1\ldots n}$ and semiring operators $\oplus := \veebar, \otimes := \wedge$. 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 $\mathrm{GF}(2^n)$, as shown in Eq. 4.

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

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

This sequence has *full periodicity*, in other words, for all $i, j \in [0, 2^n), \mathbf{S}_i = \mathbf{S}_j \implies i = j$. To uniformly sample $\boldsymbol{\sigma} \sim \Sigma^n$ without replacement, we form an injection $GF(2^n) \rightharpoonup \Sigma^d$, cycle through $\mathbf{S}$, 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, in order to sample $\sigma \sim \Sigma^2 = \{A, B, C\}^2$, we could use the primitive polynomial $x^4 + x^3 + 1$ shown below:

i	0	1	2	3	4	5	6	7
$\mathbf{S}_i$	1000	0100	0010	1001	1100	0110	1011	0101
σ	C A	B A	A C	C B		B C		B B

We will use this technique to lazily sample from the space of hole configurations without replacement as described in §6.

3.2 Encoding CFG parsing as SAT solving

By allowing the matrix $\mathbf{M}^*$ in Eq. 2 to contain bitvector variables representing holes in the string and nonterminal sets, we obtain a set of multilinear SAT equations whose solutions exactly correspond to the set of admissible repairs and their corresponding parse forests. Specifically, the repairs coincide with holes in the superdiagonal $\mathbf{M}^*_{r+1=c}$, and the parse forests occur along the upper-triangular entries $\mathbf{M}^*_{r+1<c}$.

$$\mathbf{M}^* = \begin{pmatrix} \varnothing & \{V\}_{\sigma_1} & \mathcal{L}_{1,3} & \mathcal{L}_{1,3} & \mathcal{V}_{1,4} & \cdots & \mathcal{V}_{1,n} \\ & & \{V\}_{\sigma_2} & \mathcal{L}_{2,3} & & & \\ & & & \{V\}_{\sigma_3} & & & \\ & & & & \mathcal{V}_{4,4} & & \\ & & & & & & \\ & & & & & & \mathcal{V}_{n,n} \\ \varnothing & & & & & & \varnothing \end{pmatrix}$$

Depicted above is a SAT tensor representing $\sigma_1\ \sigma_2\ \sigma_3\ \_\ \cdots\ \_$ where 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.

3.3 Deletion, substitution, and insertion

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

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

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

To generate the sketch templates, we substitute two holes at an index-to-be-repaired, $H(\sigma, i) = \sigma_{1\ldots i-1}\ \_\ \_\ \sigma_{i+1\ldots n}$, then invoke the SAT solver. 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. This procedure is repeated for all indices in the replacement set. The solutions returned by the SAT solver will be strictly equivalent to handling each edit operation as separate cases.

4 Error Recovery

Unlike classical parsers which need special care to recover from errors, if the input string does not parse, Tidyparse can return partial subtrees. If no solution exists, the upper triangular entries will appear as a jagged-shaped ridge whose peaks represent the roots of parsable ASTs. These 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. As depicted in Fig. 1, we traverse the peaks by decreasing elevation to collect partial AST branches.

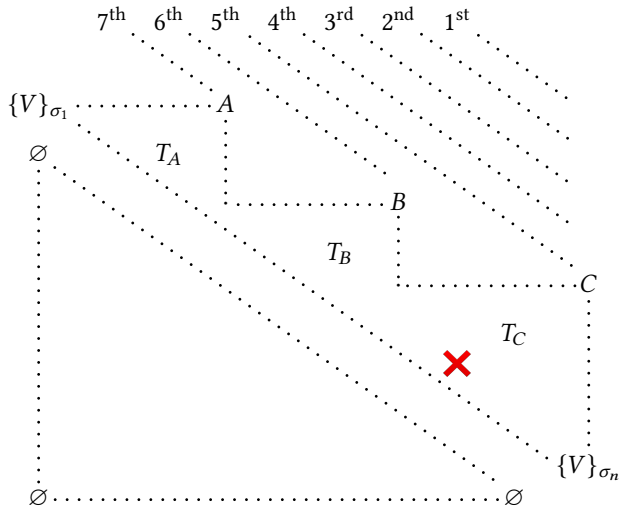
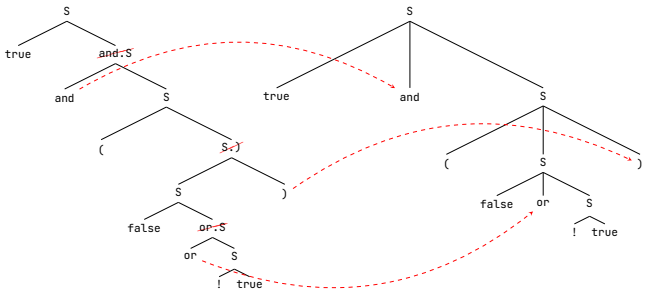


Figure 1. Peaks along the UT matrix ridge correspond to maximally parseable substrings. By recursing over upper diagonals of decreasing elevation and discarding all subtrees that fall under the shadow of another's canopy, we can recover the partial subtrees. The example depicted above contains three such branches, rooted at nonterminals C, B, A.

5 Tree Denormalization

Our parser emits a binary forest consisting of parse trees for the candidate string according to the CNF grammar, however this forest contains many so-called *Krummholz*, or *flag trees*, often found clinging to windy ridges and mountainsides.



Algorithm 1 Rewrite procedure for tree denormalization

```
procedure DENORMALIZE(t: Tree)
    stems ← { DENORMALIZE(c) | c ∈ t.children }
    if t.root ∈ V_G' \ V_G then
        return stems        ▷ Drop synthetic nonterminals.
    else                    ▷ Graft the denormalized children on root.
        return { Tree(root, stems) }
    end if
end procedure
```

To recover a parse tree congruent with the user-specified grammar, we prune all synthetic nodes and graft their stems onto the grandparent via a simple recursive procedure (Alg. 1).

6 Realtime Error Correction

Now that we have a reliable method to fix *localized* errors, $S : \mathcal{G} \times (\Sigma \cup \{\varepsilon, \_\})^n \to \{\Sigma^n\} \subseteq \mathcal{L}_{\mathcal{G}}$, given an input string, Σ^n, where should we put the holes? Assuming k holes, there are $\binom{n}{k}$ possible hole configurations (HCs), each with $(|\Sigma| + 1)^{2k}$ possible repairs (before parsing, cf. Eqs. 5-9). In practice, depending on n and k, this space can be intractable to search through exhaustively, so to facilitate real-time assistance we prioritize likely repairs according to an eight-step procedure:

1. Fetch the most recent CFG and string from the editor.
2. Exclude parsable substrings from hollowing.
3. Lazily enumerate all HCs of increasing length.
4. Sample HCs without replacement using Eq. 4.
5. Prioritize HCs first by distance to caret index, then by Earthmover's distance to a set of suspicious indices.*
6. Translate HCs to sketch templates using §3.3.
7. Feed sketch templates to an incremental SAT solver.
8. Decode and rerank models by Levenshtein distance.

* These locations can be supplied by local edit history or using tokenwise perplexity from a neural language model. Once a new repair is discovered, it is immediately displayed. Incoming keystrokes interrupt and reset the solving process.

7 Practical Example

Tidyparse requires a grammar – this can be either provided by the user or ingested from a BNF-like specification. The following is a slightly more complex grammar, designed to resemble a more realistic use case:

```
S -> A | V | ( X , X ) | X X | ( X )
A -> Fun | F | L | L in X
Fun -> fun V `->` X
F -> if X then X else X
L -> let V = X | let rec V = X
V -> Vexp | ( Vexp ) | Vexp Vexp
Vexp -> VarName | FunName | Vexp VO Vexp
Vexp -> ( VarName , VarName ) | Vexp Vexp
VarName -> a | b | c | d | e | ... | z
FunName -> foldright | map | filter
VO ->  + | - | * | / | > | = | < | `||` | &&
---
let curry f = ( fun x y -> f ( _ _ _ ) )
```

```
let curry f = ( fun x y -> f ( <X> ) )
let curry f = ( fun x y -> f ( <FunName> ) )
let curry f = ( fun x y -> f ( curry <X> ) )
...
```

7.1 Grammar Assistance

Tidyparse uses a CFG to parse the CFG, so it can provide assistance while the user is designing the CFG. For example, if the CFG does not parse, it will suggest possible fixes. In the future, we intend to use this functionality to perform example-based codesign and grammar induction.

```
B -> true | false |
```

```
B -> true | false
B -> true | false <RHS>
B -> true | false | <RHS>
...
```

7.2 Interactive Nonterminal Expansion

Users can interactively build up a complex expression by placing the caret over a placeholder they wish to expand,

```
if <Vexp> X then <Vexp> else <Vexp>
```

then invoking Tidyparse by pressing ctrl + Space, to receive a list of expressions consistient with the grammar:

```
if map X then <Vexp> else <Vexp>
if uncurry X then <Vexp> else <Vexp>
if foldright X then <Vexp> else <Vexp>
...
```

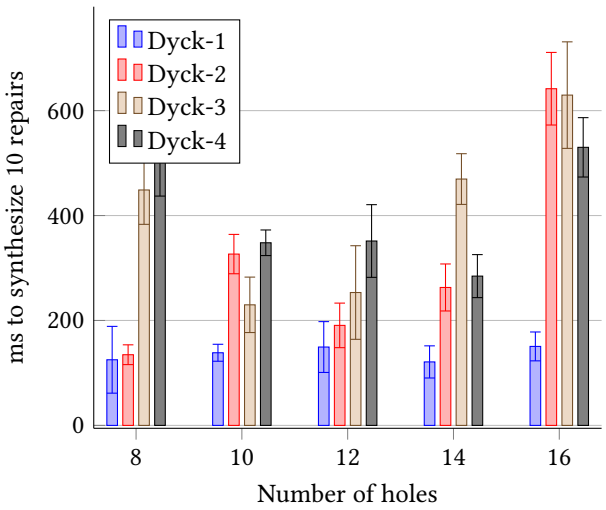
8 Latency Benchmark

In the following benchmarks, we measure the wall clock time required to synthesize solutions to length-50 strings sampled from various Dyck languages, where Dyck-n is the Dyck language containing n types of balanced parentheses.

```
D1 -> () | ( D1 ) | D1 D1
D2 -> D1 | [ ] | ( D2 ) | [ D2 ] | D2 D2
D3 -> D2 | { } | ( D3 ) | [ D3 ] | { D3 } | D3 D3
```

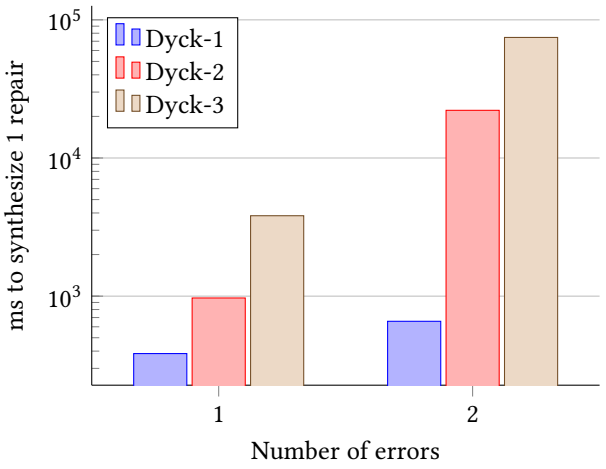
In the first experiment, we sample a random valid string $\sigma \sim \Sigma^{50} \cap \mathcal{L}_{\text{Dyck-n}}$, then replace a fixed number tokens with holes and measure the average time taken to decode ten syntactically-admissible repairs across 100 trial runs.

Error correction time with known locations



In the second experiment, we sample a random valid string as before, but delete p tokens at random and rather than provide the location(s), ask our model to solve for both the location(s) and repair by sampling uniformly from all n-token HCs, then measure the total time required to decode the first admissible repair. Note the the logarithmic scale on the y-axis.

Error correction time with unknown locations



9 Discussion

While error correction with a few errors is tolerable, latency can vary depending on many factors including string length and grammar size. If errors are known to be concentrated in specific locations, such as the beginning or end of a string, then latency is typically below 500ms. Should errors occur uniformly at random, admissible repairs can take longer to discover, however these scenarios are unusual in our experience. We observe that errors are typically concentrated nearby historical edit locations, which can be retrieved from the IDE or version control. Further optimizations that reduce the total number of repairs checked are possible by eliminating improbable sketch templates.

Tidyparse in its current form has a number of technical shortcomings: firstly it does not incorporate any neural language modeling technology at present, an omission we hope to address in the near future. Training a language model to predict likely repair locations and rank admissible results could lead to lower overall latency and more natural repairs.

Secondly, our current method generates sketch templates using a naïve enumerative search, feeding them individually to the SAT solver, which has the tendency to duplicate prior work and introduces unnecessary thrashing. Considering recent extensions of Boolean matrix-based parsing to linear context-free rewriting systems (LCFRS) [3], it may be feasible to search through these edits within the SAT solver, leading to yet unrealized and possibly significant speedups.

Lastly and perhaps most significantly, Tidyparse does not incorporate any semantic constraints, so its repairs while syntactically admissible, are not guaranteed to be semantically valid. We note however, that it is possible to encode type-based semantic constraints into the solver and intend to explore this direction more fully in future work.

Although not intended to be a dedicated parser and we make no attempt to rigorously compare parsing latency, parsing valid strings with Tidyparse is typically competitive with classical parsing methods. Our primary motivation is to facilitate the usability and explainability of parsing with errors. We envision three primary use cases: (1) helping novice programmers become more quickly familiar with a new programming language (2) autocorrecting common typos among proficient but forgetful programmers and (3) as a prototyping tool for PL educators and designers.

Featuring a grammar editor and built-in SAT solver, Tidyparse helps developers navigate the language design space, visualize syntax trees, debug parsing errors and quickly generate simple examples and counterexamples for testing. Although the algorithm may seem esoteric at first glance, in our experience it is much more interpretable than classical parsers, which exhibit poor error-recovery and diagnostics.

10 Conclusion

Tidyparse accepts a CFG and a string to parse. If the string is valid, it returns the parse forest, otherwise, it returns a set of repairs, ordered by their Levenshtein edit distance to the invalid string. Our method compiles each CFG and candidate string onto a matrix dynamical system using an extended version of Valiant's construction and solves for its fixedpoints using an incremental SAT solver. This approach to parsing has many advantages, enabling us to repair syntax errors, correct typos and generate parse trees for incomplete strings. By allowing the string to contain holes, repairs can contain either concrete tokens or nonterminals, which can be manually expanded by the user or a neural-guided search procedure. From a theoretical standpoint, this technique is particularly amenable to neural program synthesis and repair, naturally integrating with the masked-language-modeling task (MLM) used by transformer-based neural language models.

From a practical standpoint, we have implemented our approach as an IDE plugin and demonstrated its viability as a tool for live programming. Tidyparse is capable of generating repairs for invalid code in a range of toy languages. We plan to continue expanding its grammar and autocorrection functionality to cover a broader range of languages and hope to conduct a more thorough user study to validate its effectiveness in the near future. Further examples can be found at our GitHub repository: https://github.com/breandan/tidyparse

11 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, Zhixin Xiong for contributing the OCaml grammar and collaborator Nghi Bui at FPT Software for early feedback on the IDE 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] Shay B Cohen and Daniel Gildea. 2016. Parsing linear context-free rewriting systems with fast matrix multiplication. Computational Linguistics 42, 3 (2016), 421–455.

[4] 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

[5] 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