Faster by Design: Optimising UAR Sampling for Grammar Inference

Antriksh Dhand
Supervised by Dr Rahul Gopinath

School of Computer Science
The University of Sydney

February 9th, 2024

Table of Contents

Table of Contents

A brief introduction

Imagine you're in a new country and all the signs are in a foreign language. The act of slowly picking up words and patterns from these signs is analagous to grammar inference.

A brief introduction

Definition (Grammar Inference)

The process of automatically learning or inferring the underlying rules and structure of a formal grammar from a set of observed strings.

What is a formal grammar?

Using a formal grammar, we can define a (very) small subset of all the possible strings in the English language.

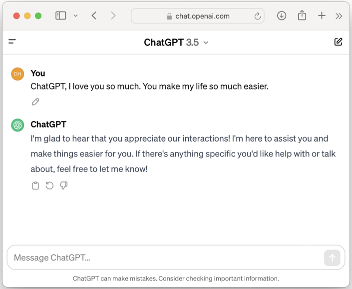
English v2.0

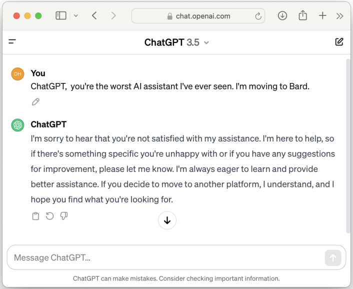
```
<sentence> ::= <noun_phrase> <verb>
<noun_phrase> ::= <article> <noun>
<article> ::= "a" | "the"
<noun> ::= "horse" | "dog" | "hamster"
<verb> ::= "stands" | "walks" | "jumps"
```

For example: `"A horse stands"` is a valid string in our language.

Applications of GI

Sentiment analysis





Grammar inference is used in sentiment analysis to help extract the grammatical structures or patterns associated with positive or negative sentiments in reviews.

Grammar inference: a broad field

Machine Learning Approaches for Grammar Inference

Automata and Regular Languages

Grammar Inference in Natural Language Processing

Algorithmic Techniques for Grammar Inference

Grammar Inference

Adaptive Learning Algorithms

Context-Free Grammars and Parsing

Probabilistic Context-Free Grammars (PCFG)

Grammar Inference in Code Analysis

Choosing the best GI algorithm

There are many grammar inference algorithms out there:

- Angluin L* + extensions
- TTT algorithm
- ARVADA
- GLADE
- ...

What measure do we use to say that one grammar inference algorithm is better than another?

Quantifying "better"

Let the original grammar (the one we are trying to infer) be denoted G. Let the output of the grammar inference algorithm be G'.

Fact

A grammar inference algorithm is "good" if G' is similar to G.

Quantifying "better"

We measure similarity using two measures:

Definition (Precision)

If we run a generating fuzzer on G', how close is the output to running a parser on G?

Definition (Recall)

If we run a generating fuzzer on G, how close is the output to running a parser on G'?

Note: the generation must be uniform at random.
i.e. when sampling a string s from a grammar, $P(s) = c, c \in \mathbb{R}$.

Quantifying "better"

We combine Precision and Recall using the F1 score:

Definition

$$\text{F1 score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

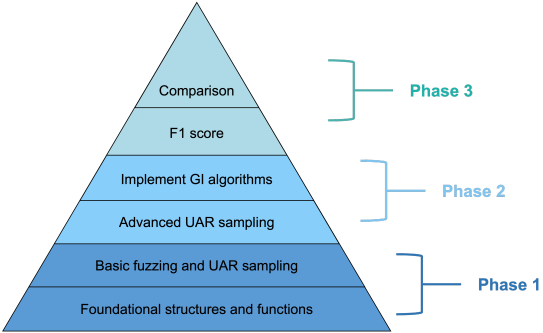
Comparing effectiveness of grammar inference algorithms is then equivalent to comparing F1 scores.

Broad project goal

Set up a toolchain to compare grammar inference algorithms.

A pathway to comparison

The pathway to comparison is extensive and involves multiple stages. We've designed a phased approach to address the complexity of the project.



Phase 1: Problem statement

Challenge

Many current algorithms for UAR sampling are written in Python which is a slower, interpreted language.

- Would ChatGPT be as successful if each query took 5 minutes to run?
- In practice, efficiency is key.
- To pave the way for future advancements, we must utilise a high-performance language and optimise the code for efficiency.

Phase 1: Goals

1. Build more efficient tools, structures, and functions to work with grammars.
2. Implement UAR sampling using this foundation.

Table of Contents

Why C

- C is a compiled language, leading to faster execution times compared to interpreted languages like Python.
- C offers manual memory management which allows us control over using dynamic or static allocation, enhancing efficiency and performance.

Developing foundations

Set up a robust and optimised method of storing a grammar in C.

Instead of using strings in our representation, we map each token in the grammar to an 8-bit number.

- Non-terminals are assigned 8-bit keys starting from `0x80`
- Terminals are assigned 8-bit keys starting from `0x00`

This allows us to:

- Check if a token is non-terminal in $O(1)$ time (MSB)
- Greatly improve the space complexity of our algorithms

Developing foundations

```
<expression> ::= <term> "+" <term>
<term> ::= <factor> "*" <factor>
<factor> ::= "0" | "1" | "2" | ... | "9"
```

The above grammar would be converted to:

```
0x80 ::= 0x81 0x00 0x81
0x81 ::= 0x82 0x01 0x82
0x82 ::= 0x02 | 0x03 | 0x04 | ... | 0x0B
```

Developing foundations

We also store tokens in our representation in the order they appear in the grammar i.e. grammars are stored as such:

```
GRAMMAR = [<NT-struct for 0x80>, <NT-struct for 0x81>,
<NT-struct for 0x82>, ...]
```

- This allows us $O(1)$ time index lookup of each non-terminal in the grammar (4 least significant bits)

Allowing “plug-and-play” functionality

Develop a program to convert existing grammar to C representation.

As most researchers use JSON to represent their grammars, we developed a JSON-to-C converter to jumpstart the development process for users.

Allowing "plug-and-play" functionality

JSON

```
{
    "<start>": [["<sentence>"]],
    "<sentence>": [["<noun_phrase>", "<verb>"]],
    "<noun_phrase>": [["<article>", "<noun>"]],
    "<noun>": [["horse"], ["dog"], ["hamster"]],
    "<article>": [["a"], ["the"]],
    "<verb>": [["stands"], ["walks"], ["jumps"]]
}
```

C

```
Grammar GRAMMAR = {
  6,
  {
    {
      // <start>
      0x80,
      1,
      {
        {
          // <sentence>
          1,
          {0x81}
        }
      }
    },
    {
      // <sentence>
      0x81,
      1,
      {
        {
          // <noun_phrase>, <verb>
          2,
          {0x82, 0x83}
          ...
```

UAR sampling implementation

- Fuzzers' string sampling may lack uniformity due to their reliance on grammar structure.
- Our approach generates all potential strings from the grammar.
- We then employ random sampling from the exhaustive set of strings, ensuring uniform distribution.
- This method guarantees robust and reliable sampling, overcoming the limitations of conventional fuzzers.

UAR sampling implementation

Use efficient data structures to implement sampling algorithm.

We implemented custom hash tables in C to memoize the grammar's structure.

- `KeyNode` and `RuleNode` objects were created in C to represent Keys and Rules in the grammar.
- The memoization process involves recursively traversing the grammar rules and storing these objects in hash tables.
- By storing previously computed results, redundant computations are avoided, optimising the sampling process.

UAR sampling implementation

We utilised Jenkins' one-at-a-time hash function to try and obtain a uniform spread of keys in the hash tables and minimise collisions.

```
uint32_t hash = 0;

for (size_t i = 0; i < 8; ++i) {
    hash += key & 1;
    hash += (hash << 10);
    hash ^= (hash >> 6);
    key >>= 1;
}

hash += (hash << 3);
hash ^= (hash >> 11);
hash += (hash << 15);
```

Table of Contents

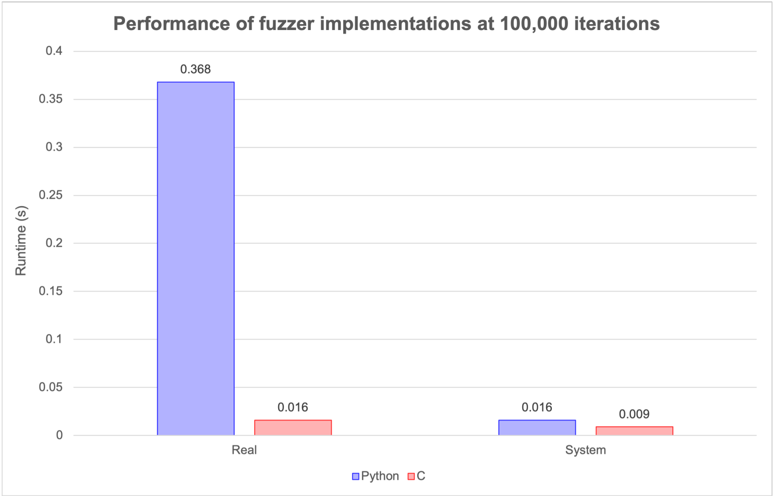
Performance improvements

Our C implementation of a basic fuzzer was up to

23x faster

than its Python counterpart.

Performance improvements



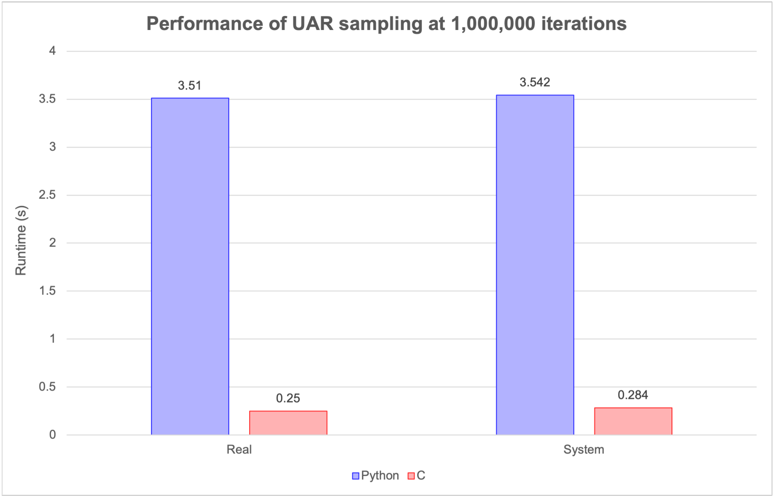
Performance improvements

Our C implementation of the UAR sampling algorithm ran up to

14x faster

than its Python counterpart.

Performance improvements

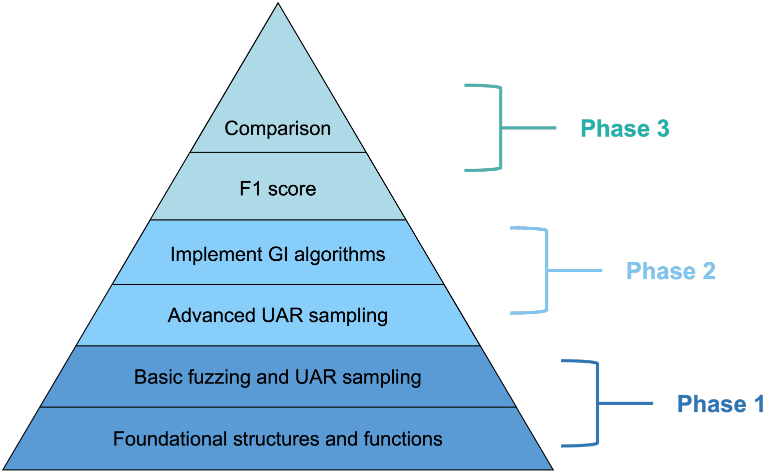


Contribution to the field

- Our findings not only substantiate the foundational work we have done in optimising our grammar structure but also provide a valuable and accessible asset to the programming community.
- The transparent documentation ensures accessibility and encourages collaborative exploration in the field of grammar inference.

Table of Contents

Future work


Comparison
F1 score
Implement GI algorithms
Advanced UAR sampling
Basic fuzzing and UAR sampling
Foundational structures and functions
Phase 3
Phase 2
Phase 1

Resources

The code produced as a result of this research project is now a well-documented open-source C library and can be accessed at github.com/antrikshdhand/gfuzztools.

Acknowledgements

I am sincerely grateful for the time and effort invested by Dr. Rahul Gopinath in sharing his expertise, providing valuable suggestions, and serving as an inspiring mentor throughout this project.

Thank You

Contact information

- Email: adha5655@uni.sydney.edu.au
- GitHub: github.com/antrikshdhand
- LinkedIn: linkedin.com/in/antrikshdhand

References and further reading

R. Gopinath.
Uniform Random Sampling of Strings from Context-Free Grammar
rahul.gopinath.org, 2021.

C. Higuera.
Grammatical Inference: Learning Automata and Grammars
Cambridge University Press, 2010.

D. Angluin.
Learning Regular Sets from Queries and Counterexamples
Information and Computation, 2(75):87–106, 1982.