Uniform Random Sampling of Strings from Context-Free Grammar

Published: Jul 27, 2021

Note: The algorithm discussed in this post is only useful for nonambiguous context-free grammars without epsilon (empty expansion nonterminal).

In the previous post I talked about how to generate input strings from any given context-free grammar. While that algorithm is quite useful for fuzzing, one of the problems with that algorithm is that the strings produced from that grammar is skewed toward shallow strings.

For example, consider this grammar:

Contents

Important: Pyodide takes time to initialize. Initialization completion is indicated by a red border around *Run all* button.

Run all

```
G = {
    "<start>" : [["<digits>"]],
    "<digits>" : [["<digit>", "<digits>"],
        ["<digit>"]],
    "<digit>" : [[str(i)] for i in range(10)]
}
```

Run

To generate inputs, let us load the limit fuzzer from the previous post.

▸ Available Packages

```
import simplefuzzer as fuzzer
```

Run

The generated strings (which generate random integers) are as follows

```
gf = fuzzer.LimitFuzzer(G)
for i in range(10):
    print(gf.fuzz('<start>'))
```

Run

```
7
060
5
2
1
8018
9
42
93623
67934639
```

As you can see, there are more single digits in the output than longer integers. Almost half of the generated strings are single character. If we modify our grammar as below

```
G2 = {
    "<start>" : [["<digits>"]],
    "<digits>" : [["<digit>", "<digits>"],
                  ["<digit>"],
                  ["<threedigits>"],
                  ],
    "<digit>" : [[str(i)] for i in range(10)],
     "<threedigits>" : [[str(i) + str(j) + str(k)] for i in range(2) for j
in range(2) for k in range(2)]
}
```

Run

and run it again

```
gf = fuzzer.LimitFuzzer(G2)
for i in range(10):
    print(gf.fuzz('<start>'))
```

Run

```
26667
177
001
011
1
8111
2111
9110
000
71
```

you will notice a lot more three digit wide binary numbers are produced. In fact, now, more than one third of the generated strings are likely to be three digits. This is because of the way the grammar is written. That is, there are three possible expansions of the `<digits>` nonterminal, and our fuzzer chooses one of the expansions with equal probability. This leads to the skew in the inputs we see above. This is interesting, but why is it a problem for practitioners? Consider these two grammars.

```
E1 = {
 '<start>': [['<E>']],
 '<E>': [['<F>', '*', '<E>'],
         ['<F>', '/', '<E>'],
         ['<F>']],
 '<F>': [['<T>', '+', '<F>'],
         ['<T>', '-', '<F>'],
         ['<T>']],
 '<T>': [['(', '<E>', ')'],
         ['<D>']],
 '<D>': [['0'], ['1']]
}
E2 = {
 '<start>': [['<E>']],
 '<E>': [['<T>', '+', '<E>'],
            ['<T>', '-', '<E>'],
            ['<T>']],
 '<T>': [['<F>', '*', '<T>'],
            ['<F>', '/', '<T>'],
            ['<F>']],
 '<F>': [['(', '<E>', ')'],
         ['<D>']],
 '<D>': [['0'], ['1']]
}
```

Run

Now, let us look at the generated strings. The first:

```
print("E1")
e1f = fuzzer.LimitFuzzer(E1)
for i in range(10):
    print(e1f.fuzz('<start>', max_depth=1))
```

Run

```
E1
```

```
0*0
1
1
1
1*0
0/0
1/0
1*0
0*0
0*0
```

And the second:

```python
print("E2")
e2f = fuzzer.LimitFuzzer(E2)
for i in range(10):
    print(e2f.fuzz('<start>', max_depth=1))
```

Run

```
E2
0-1
1+1
1
1+1
1+0
0-1
1-1
1-1
1
1
```

Notice that both grammars describe exactly the same language, but the first one has a higher proportion of multiplications and divisions while the second one has a higher proportion of additions and subtractions. This means that the effectiveness of our fuzzers is determined to a large extent by the way the context-free grammar is written. Another case is when one wants to compare the agreement between two grammars. I talked about this before. As you can see from the above cases, the same language can be described by different grammars, and it is undecidable in general whether two context-free grammars describe the same language [1]. So, we will have to go for statistical means. To start with, we need to be able to randomly sample from the strings that can be produced from the grammar. So, the minimal requirement is as follows:

- We need to be able to randomly sample a string that can be generated from the grammar. To make this happen, let us split this into two simpler requirements:
- We can find the number of strings of a given size that can be produced from the grammar.
- We can enumerate the strings that can be produced from the grammar, and pick a specific string given its index in the enumeration.

Once we have both these abilities, then we can combine them to provide random sampling of derived strings. So, let us see how to achieve that.

A Naive Implementation.

Counting the number of strings

Let us first implement a way to count the number of strings that can be possibly generated.

Count strings generated by a nonterminal

We first implement `key_get_num_strings()` to count the number of strings of a given size `l_str` generated by a token. For finding the number we first check if the given token is a `terminal` symbol. If it is, then there is only one choice. That symbol is the string. The constraint is that the length of the string should be as given by `l_str`. if not, the total number of strings that can be generated from this token is the total number of strings we can generated from each individual expansion.

```
def key_get_num_strings(key, grammar, l_str):
    if not fuzzer.is_nonterminal(key):
        if l_str == len(key): return 1
        else: return 0
    s = 0
    rules = grammar[key]
    for rule in rules:
        s += rule_get_num_strings(rule, grammar, l_str)
    return s
```

Run

Let us test this out, but only with a token

```
count = key_get_num_strings('1', G, 1)
print("len:", count)
count = key_get_num_strings('1', G, 2)
print("len:", count)
count = key_get_num_strings('2', G, 1)
print("len:", count)
```

Run

```
len: 1
len: 0
len: 1
```

Count strings generated by a rule

Next, we implement `rule_get_num_strings()` which counts the number of strings of given size `l_str` that can be generated by a rule (an expansion of a nonterminal). Here, we treat each rule as a head followed by a tail. The token is the first symbol in the rule. The idea is that, the total number of strings that can be generated from a rule is the multiplication of the number of strings that can be generated from the head by the total strings that can be generated by the tail.

We need to handle the base case when there is no tail. In that case, we return the number of strings for the head.

The complication is that we want to generate a specific size string. So, we split that size (`l_str`) between the head and tail and count strings generated by each possible split.

```python
def rule_get_num_strings(rule, grammar, l_str):
    if not rule: return 0

    token, *tail = rule
    if not tail:
        return key_get_num_strings(token, grammar, l_str)

    sum_rule = 0
    for partition in range(1, l_str+1): # inclusive
        h_len, t_len = partition, l_str - partition
        s_in_h = key_get_num_strings(token, grammar, h_len)
        if s_in_h == 0: continue

        s_in_t = rule_get_num_strings(tail, grammar, t_len)
        if s_in_t == 0: continue

        sum_rule += s_in_h * s_in_t
    return sum_rule
```

Run

Using it.

```python
count = key_get_num_strings('<start>', G, 2)
print("len:", count)
```

Run

```
len: 100
```

Generation of strings

Let us next implement a way to generate all strings of a given size. Here, in `key_get_strings()`, we pass in the key, the grammar and the length of the string expected. Note the symmetry to `key_get_num_strings()`.

For generating a string from a key, we first check if it is a `terminal` symbol. If it is, then there is only one choice. That symbol is the string. The constraint is that the length of the string should be as given by `l_str`. if not, then we find all the expansion rules of the corresponding definition and generate strings from each expansion of the given size `l_str`; the concatenation of which is the required string list.

```python
def key_get_strings(key, grammar, l_str):
    if not fuzzer.is_nonterminal(key):
        if l_str == len(key): return [key]
        else: return []
    s = []
    rules = grammar[key]
    for rule in rules:
        s_ = rule_get_strings(rule, grammar, l_str)
        s.extend(s_)
    return s
```

Run

Next, we come to the rule implementation given by `rule_get_strings()` Here, we treat each rule as a head followed by a tail. The token is the first symbol in the rule. The idea is that, the strings that are generated from this rule will have one of the strings generated from the token followed by one of the strings generated from the rest of the rule. This also provides us with the base case. If the rule is empty, we are done. if it is not the base case, then we first extract the strings from the token head, then extract the strings from the tail, and concatenate them pairwise.

Note the symmetry to `rule_get_num_strings()`

The complication here is the number of characters expected in the string. We can divide the number of characters — `l_str` between the head and the tail. That is, if the string from head takes up `x` characters, then we can only have `l_str - x` characters in the tail. To handle this, we produce a loop with all possible splits between head and tail. Of course not all possible splits may be satisfiable. Whenever we detect an impossible split — by noticing that `s_` is empty, we skip the loop.

```
def rule_get_strings(rule, grammar, l_str):
    if not rule: return []

    token, *tail = rule
    if not tail:
        return key_get_strings(token, grammar, l_str)

    sum_rule = []
    for partition in range(1,l_str+1): # inclusive
        h_len, t_len = partition, l_str - partition
        s_in_h = key_get_strings(token, grammar, h_len)
        if not s_in_h: continue

        s_in_t = rule_get_strings(tail, grammar, t_len)
        if not s_in_t: continue

        for sh in s_in_h:
            for st in s_in_t:
                sum_rule.append(sh + st)

    return sum_rule
```

Run

Using it.

```
strings = key_get_strings('<start>', G, 2)
print("len:", len(strings))
print(strings)
```

Run

```
len: 100
['00', '01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12',
'13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24', '25',
'26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36', '37', '38',
'39', '40', '41', '42', '43', '44', '45', '46', '47', '48', '49', '50', '51',
'52', '53', '54', '55', '56', '57', '58', '59', '60', '61', '62', '63', '64',
'65', '66', '67', '68', '69', '70', '71', '72', '73', '74', '75', '76', '77',
'78', '79', '80', '81', '82', '83', '84', '85', '86', '87', '88', '89', '90',
'91', '92', '93', '94', '95', '96', '97', '98', '99']
```

The problem with these implementations is that it is horribly naive. Each call recomputes the whole set of strings or the count again and again. However, many nonterminals are reused again and again, which means that we should be sharing the results. Let us see how we can memoize the results of these calls.

A Memoized Implementation.

Counting the number of strings

We first define a data structure to keep the key nodes. Such nodes help us to identify the corresponding rules of the given key that generates strings of `l_str` size.

```
class KeyNode:
    def __init__(self, token, l_str, count, rules):
        self.token = token
        self.l_str = l_str
        self.count = count
        self.rules = rules

    def __str__(self):
          return "key: %s <%d> count:%d" % (repr(self.token), self.l_str,
self.count)

    def __repr__(self):
          return "key: %s <%d> count:%d" % (repr(self.token), self.l_str,
self.count)
```

Run

We also define a data structure to keep the rule nodes. Such rules contain both the head `token` as well as the `tail`, the `l_str` as well as the count

```
class RuleNode:
    def __init__(self, key, tail, l_str, count):
        self.key = key
        self.tail = tail
        self.l_str = l_str
        self.count = count
        assert count

    def __str__(self):
                return "head: %s tail: (%s) <%d> count:%d" %
(repr(self.key.token), repr(self.tail), self.l_str, self.count)

    def __repr__(self):
                return "head: %s tail: (%s) <%d> count:%d" %
(repr(self.key.token), repr(self.tail), self.l_str, self.count)
```

Run

globals

```
rule_strs = { }

key_strs = { }

EmptyKey = KeyNode(token=None, l_str=None, count=0, rules = None)
```

Run

Populating the linked data structure.

This follows the same skeleton as our previous functions. First the keys

```
def key_get_def(key, grammar, l_str):
    if (key, l_str) in key_strs: return key_strs[(key, l_str)]

    if not fuzzer.is_nonterminal(key):
        if l_str == len(key):
            key_strs[(key, l_str)] = KeyNode(
                    token=key, l_str=l_str, count=1, rules = [])
            return key_strs[(key, l_str)]
        else:
            key_strs[(key, l_str)] = EmptyKey
            return key_strs[(key, l_str)]
    s = []
    count = 0
    rules = grammar[key]
    for rule in rules:
        s_ = rules_get_def(rule, grammar, l_str)
        count += sum([_.count for _ in s_])
        s.extend(s_)
     key_strs[(key, l_str)] = KeyNode(token=key, l_str=l_str, count=count,
rules=s)
    return key_strs[(key, l_str)]
```

Run

Now the rules. The complication from before is that, if the count is zero, we do not return an array with a zero rulenode. Instead we return an empty array.

```
def rules_get_def(rule_, grammar, l_str):
    rule = tuple(rule_)
    if not rule: return []
    if (rule, l_str) in rule_strs: return rule_strs[(rule, l_str)]

    token, *tail = rule
    if not tail:
        s_ = key_get_def(token, grammar, l_str)
        if not s_.count: return []
        return [RuleNode(key=s_, tail=[], l_str=l_str, count=s_.count)]

    sum_rule = []
    count = 0
    for partition in range(1, l_str+1):
        h_len, t_len = partition, l_str - partition
        s_in_h = key_get_def(token, grammar, h_len)
        if not s_in_h.count: continue

        s_in_t = rules_get_def(tail, grammar, t_len)
        if not s_in_t: continue

        count_ = 0
        for r in s_in_t:
            count_ += s_in_h.count * r.count

        if not count_: continue

        count += count_
              rn   =   RuleNode(key=s_in_h,   tail=s_in_t,   l_str=partition,
count=count_)
        sum_rule.append(rn)

    rule_strs[(rule, l_str)] = sum_rule
    return rule_strs[(rule, l_str)]
```

Run

Using it.

```
key_node = key_get_def('<start>', G, 2)
print("len:", key_node.count)
```

Run

```
len: 100
```

We can of course extract the same things from this data structure.

Count

For example, if we wanted to recompute counts without using the `count` attribute

Key Count

For the keys

```
def key_get_count(key_node):
    if not key_node.rules: return 1
    slen = 0
    for rule in key_node.rules:
        s = rule_get_count(rule)
        slen += s
    return slen
```

Run

Rule Count

For the rules

```
def rule_get_count(rule_node):
    slen = 0
    s_k = key_get_count(rule_node.key)
    if not rule_node.tail: return s_k

    for rule in rule_node.tail:
        s_t = rule_get_count(rule)
        slen = s_k * s_t

    return slen
```

Run

Using it.

```
count = key_get_count(key_node)
print("len:", count)
```

Run

```
len: 100
```

Strings

For example, if we wanted to compute strings

Key Strings

```
def key_extract_strings(key_node):
    # key node has a set of rules
    if not key_node.rules: return [key_node.token]
    strings = []
    for rule in key_node.rules:
        s = rule_extract_strings(rule)
        if s:
            strings.extend(s)
    return strings
```

Run

Rule Strings

```
def rule_extract_strings(rule_node):
    s_h = key_extract_strings(rule_node.key)
    if not rule_node.tail: return s_h

    strings = []
    for rule in rule_node.tail:
        s_t = rule_extract_strings(rule)
        for s1 in s_h:
            for s2 in s_t:
                strings.append(s1 + s2)
    return strings
```

Run

Using it.

```
strings = key_extract_strings(key_node)
print("len:", len(strings))
```

Run

```
len: 100
```

Random Access

But more usefully, we can now use it to randomly access any particular string. The idea is same as before. If the index being requeted is within the strings of the node expansion, return it. Any given nonterminal may be either a terminal symbol or it may be expanded by one or more rules.

In the casee of a terminal symbol (no rules), we have no choice, but to reutrn the token. (We should probably `assert at == 0`). But in the case of nonterminal symbol, we can pass the request to the specifc rule that has the requested index.

At Keys

```
def key_get_string_at(key_node, at):
    assert at < key_node.count
    if not key_node.rules: return key_node.token

    at_ = 0
    for rule in key_node.rules:
        if at < (at_ + rule.count):
            return rule_get_string_at(rule, at - at_)
        else:
            at_ += rule.count
    return None
```

Run

At Rules

In the case of rules, the idea is mostly the same as before. If there is no tail, we get the base case.

In case there is a tail, we split the rule into a head and a tail. Note that a single rule node corresponds to a specific partition between the head and tail. That is, the head and tails in the rule node are compatible with each other in terms of length. That is, we do not have to worry about partitions.

The total number of strings is `num(strings in head) x num(strings in tail)`. That is, for each string that correspond to the head, there is a set of tails. So, to get a string at a particular index, we need to iterate through each previous string in the head, multiplied by the number of strings in the tail. The count of such strings in head is given by `len_s_h`, and each head is indexed by `head_idx`. Then, we keep appending the number of strings in the rule tail. When the count reaches a given head, we identify the corresponding head by head_idx, and extract the corresponding string in the tail.

```
def rule_get_string_at(rule_node, at):
    assert at < rule_node.count
    if not rule_node.tail:
        s_k = key_get_string_at(rule_node.key, at)
        return s_k

    at_ = 0
    len_s_h = rule_node.key.count
    for rule in rule_node.tail:
        for head_idx in range(len_s_h):
            if at < (at_ + rule.count):
                s_k = key_get_string_at(rule_node.key, head_idx)
                return s_k + rule_get_string_at(rule, at - at_)
            else:
                at_ += rule.count
    return None
```

Run

Using it.

```
at = 3
strings = key_extract_strings(key_node)
print("strting[%d]" % at, repr(strings[at]))

string = key_get_string_at(key_node, at)
print(repr(string))
```

Run

```
strting[3] '03'
'03'
```

Random Sampling

Once we have random access of a given string, we can turn it to random sampling.

```
import random
```

Run

```
key_node_g = key_get_def('<start>', E1, 5)
print(key_node_g.count)
at = random.randint(0,key_node_g.count)
print('at:', at)
strings = key_extract_strings(key_node_g)
print("strting[%d]" % at, repr(strings[at]))
string = key_get_string_at(key_node_g, at)
print(repr(string))
```

Run

```
178
at: 146
strting[146] '1-0+0'
'1-0+0'
```

This is random sampling from restricted set — the set of derivation strings of a given length. How do we extend this to lengths up to a given length? The idea is that for generating strings of length up to `n`, we produce and use nonterminals that generate strings of length up to `n-x` where `x` is the length of first terminals in expansions. This means that we can build the `key_node` data structures recursively from 1 to `n`, and most of the parts will be shared between the `key_node` data structures of different lengths. Another issue this algorithm has is that it fails when there is left recursion. However, it is fairly easy to solve as I showed in a previous post. The idea is that there is a maximum limit to the number of useful recursions. Frost et. al.[2] suggests a limit of `m * (1 + |s|)` where `m` is the number of nonterminals in the grammar and `|s|` is the length of input. So, we use that here for limiting the recursion.

```
import bisect

class RandomSampleCFG:
    def __init__(self, grammar):
        self.grammar = grammar
        self.rule_strs = { }
        self.key_strs = { }
         self.EmptyKey = KeyNode(token=None, l_str=None, count=0, rules =
None)
        self.ds = {}
        self.recursion_ctr = {}
        self.count_nonterminals = len(grammar.keys())
```

Run

Populating the linked data structure.

```
class RandomSampleCFG(RandomSampleCFG):
    def key_get_def(self, key, l_str):
          if (key, l_str) in self.key_strs: return self.key_strs[(key,
l_str)]

        if not fuzzer.is_nonterminal(key):
            if l_str == len(key):
                self.key_strs[(key, l_str)] = KeyNode(
                        token=key, l_str=l_str, count=1, rules = [])
                return self.key_strs[(key, l_str)]
            else:
                self.key_strs[(key, l_str)] = EmptyKey
                return self.key_strs[(key, l_str)]

        # number strings in definition = sum of number of strings in rules
        if key not in self.recursion_ctr: self.recursion_ctr[key] = 0

        self.recursion_ctr[key] += 1

        limit = self.count_nonterminals * (1 + l_str) # m * (1 + |s|)
        # remove left-recursive rules -- assumes no epsilon
        if self.recursion_ctr[key] > limit:
            rules = [r for r in self.grammar[key] if r[0] != key]
        else:
            rules = self.grammar[key] # can contain left recursion


        s = []
        count = 0
        for rule in rules:
            s_s = self.rules_get_def(rule, l_str) # returns RuleNode
            for s_ in s_s:
                assert s_.count
                count += s_.count
                s.append(s_)
        self.key_strs[(key, l_str)] = KeyNode(
                token=key, l_str=l_str, count=count, rules = s)
        return self.key_strs[(key, l_str)]

    # Now the rules.

    def rules_get_def(self, rule_, l_str):
        rule = tuple(rule_)
        if not rule: return []
         if (rule, l_str) in self.rule_strs: return self.rule_strs[(rule,
l_str)]

        token, *tail = rule
        if not tail:
            s_ = self.key_get_def(token, l_str)
            if not s_.count: return []
                       return   [RuleNode(key=s_,   tail=[],   l_str=l_str,
count=s_.count)]

        sum_rule = []
        count = 0
        for l_str_x in range(1, l_str+1):
            s_ = self.key_get_def(token, l_str_x)
            if not s_.count: continue

            rem = self.rules_get_def(tail, l_str - l_str_x)
            count_ = 0
            for r in rem:
                count_ += s_.count * r.count

            if count_:
                count += count_
                        rn   =  RuleNode(key=s_,  tail=rem,  l_str=l_str_x,
count=count_)
                sum_rule.append(rn)
        self.rule_strs[(rule, l_str)] = sum_rule
        return self.rule_strs[(rule, l_str)]

    def key_get_string_at(self, key_node, at):
        assert at < key_node.count
        if not key_node.rules: return (key_node.token, [])
        at_ = 0
        for rule in key_node.rules:
            if at < (at_ + rule.count):
                 return (key_node.token, self.rule_get_string_at(rule, at -
at_))
            else:
                at_ += rule.count
        assert False

    def rule_get_string_at(self, rule_node, at):
        assert at < rule_node.count
        if not rule_node.tail:
            s_k = self.key_get_string_at(rule_node.key, at)
            return [s_k]

        len_s_k = rule_node.key.count
        at_ = 0
        for rule in rule_node.tail:
            for i in range(len_s_k):
                if at < (at_ + rule.count):
                    s_k = self.key_get_string_at(rule_node.key, i)
                    return [s_k] + self.rule_get_string_at(rule, at - at_)
                else:
                    at_ += rule.count
        assert False

    # produce a shared key forest.
    def produce_shared_forest(self, start, upto):
        for length in range(1, upto+1):
            if length in self.ds: continue
            key_node_g = self.key_get_def(start, length)
            count = key_node_g.count
            self.ds[length] = key_node_g
        return self.ds

    def compute_cached_index(self, n, cache):
        cache.clear()
        index = 0
        for i in range(1, n+1):
            c = self.ds[i].count
            if c:
                cache[index] = self.ds[i]
                index += c
        total_count = sum([self.ds[l].count for l in self.ds if l <= n])
        assert index == total_count
        return cache

    def get_total_count(self, cache):
        last = list(cache.keys())[-1]
        return cache[last].count + last


    # randomly sample from 1 up to `l` length.
    def random_sample(self, start, l, cache=None):
        assert l > 0
        if l not in self.ds:
            self.produce_shared_forest(start, l)
        if cache is None:
            cache = self.compute_cached_index(l, {})
        total_count = self.get_total_count(cache)
        choice = random.randint(0, total_count-1)
        my_choice = choice
        # get the cache index that is closest.
        index = bisect.bisect_right(list(cache.keys()), choice)
        cindex = list(cache.keys())[index-1]
        my_choice = choice - cindex # -1
        return choice, self.key_get_string_at(cache[cindex], my_choice)

    # randomly sample n items from 1 up to `l` length.
    def random_samples(self, start, l, n):
        cache = {}
        lst = []
        for i in range(n):
            lst.append(self.random_sample(start, l, cache))
        return lst
```

Run

Using it.

```
rscfg = RandomSampleCFG(E1)
max_len = 10
rscfg.produce_shared_forest('<start>', max_len)
for i in range(10):
    at = random.randint(1, max_len) # at least 1 length
    v, tree = rscfg.random_sample('<start>', at)
    string = fuzzer.tree_to_string(tree)
    print("mystring:", repr(string), "at:", v, "upto:", at)
```

Run

```
mystring: '0' at: 0 upto: 2
mystring: '(1+1)/0' at: 1520 upto: 7
mystring: '1+1+0+0+1' at: 18421 upto: 9
mystring: '0' at: 0 upto: 2
mystring: '1+0+0*1' at: 755 upto: 8
mystring: '0*0/0' at: 24 upto: 6
mystring: '0-0' at: 14 upto: 5
mystring: '0' at: 0 upto: 2
mystring: '1+0*0-0' at: 602 upto: 8
mystring: '1/1' at: 9 upto: 3
```

What about a left-recursive grammar? Here is an example:

```
LRG = {
"<start>": [
    ["<L>"]],
"<L>": [
    ["A"],
    ["<L>","B"]]
}
```

Run

Using it.

```
rscfg = RandomSampleCFG(LRG)
max_len = 10
rscfg.produce_shared_forest('<start>', max_len)
for i in range(10):
    at = random.randint(1, max_len) # at least 1 length
    v, tree = rscfg.random_sample('<start>', at)
    string = fuzzer.tree_to_string(tree)
    print("mystring:", repr(string), "at:", v, "upto:", at)
```

Run

```
mystring: 'A' at: 0 upto: 3
mystring: 'AB' at: 1 upto: 7
mystring: 'AB' at: 1 upto: 4
mystring: 'A' at: 0 upto: 9
mystring: 'ABB' at: 2 upto: 9
mystring: 'ABBB' at: 3 upto: 5
mystring: 'A' at: 0 upto: 9
mystring: 'ABB' at: 2 upto: 9
mystring: 'ABBBB' at: 4 upto: 6
mystring: 'A' at: 0 upto: 1
```

There are a few limitations to this algorithm. The first is that it does not take into account epsilons – that is empty derivations. It can be argued that it is not that big of a concern since any context-free grammar can be made epsilon free. However, if you are not too much worried about exactness, and only want an approximate random sample, I recommend that you replace empty rules with a rule containing a special symbol. Then, produce your random sample, and remove the special symbol from the generated string. This way, you can keep the structure of the original grammar. The next limitation is bigger. This implementation does not take into account ambiguity in grammar where multiple derivation trees can result in the same string. This means that such strings will be more likely to appear than other strings. While there are a number of papers [3] [4] [5] [6] [7] that tackle the issue of statistical sampling, with better runtime and space characteristics, we are not aware of any that fixes both issues (Gore et al.[8] is notable for showing an *almost uniform random sampling* result). Bertoni et al. shows[9] that for some inherently ambiguous languages, the problem becomes intractable.

The code for this post is available here.

References

Run all

Artifacts

The runnable Python source for this notebook is available here.

The installable python wheel `cfgrandomsample` is available here.

1. Bar-Hillel, Yehoshua, Micha Perles, and Eli Shamir. “On formal properties of simple phrase structure grammars.” STUF-Language Typology and Universals 14.1-4 (1961): 143-172. ↩

2. Richard A. Frost, Rahmatullah Hafiz, and Paul C. Callaghan. Modular and efficient top-down parsing for ambiguous left recursive grammars. IWPT 2007 ↩

3. Madhavan, Ravichandhran, et al. “Automating grammar comparison.” Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2015. ↩

4. McKenzie, Bruce. “The Generation of Strings from a CFG using a Functional Language.” (1997). ↩

5. McKenzie, Bruce. “Generating strings at random from a context free grammar.” (1997). ↩

6. Harry G. Mairson. Generating words in a context-free language uniformly at random. Information Processing Letters, 49(2):95{99, 28 January 1994 ↩

7. Hickey, Timothy, and Jacques Cohen. “Uniform random generation of strings in a context-free language.” SIAM Journal on Computing 12.4 (1983): 645-655. ↩

8. Gore, Vivek, et al. “A quasi-polynomial-time algorithm for sampling words from a context-free language.” Information and Computation 134.1 (1997): 59-74. ↩

9. Bertoni, Alberto, Massimiliano Goldwurm, and Nicoletta Sabadini. “The complexity of computing the number of strings of given length in context-free languages.” Theoretical Computer Science 86.2 (1991): 325-342. ↩