Generating Strings at Random from a Context Free Grammar

Bruce McKenzie

Department of Computer Science, University of Canterbury, Christchurch, New Zealand.
E-Mail: bruce@cosc.canterbury.ac.nz

Abstract

Given a context free grammar (CFG) G and an integer $n >= 0$ we present an algorithm for generating strings derivable from the grammar of length n such that all strings of length n are equally likely. The algorithm requires a pre-processing stage which calculates the number of strings of length $k <= n$ derivable from each postfix β where $A \rightarrow \alpha\beta$ is a production from the grammar. This step requires $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space. The subsequent string generation step uses these counts to generate a string in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.

Key words: Analysis of algorithms, Context-free languages, Uniform random generation, memoization.

1 Introduction

Let $G = (N, T, P, S)$ be a CFG where N is a set of non-terminal symbols, T is a set of teminal symbols, P is a set of productions of the form $A \rightarrow \alpha$ ($A \in N$, $\alpha \in (N \cup T)^*$) and start symbol $S \in N$. This paper is concerned with the problem of generating strings at random derivable from G. Such strings can be used to test parsers and, by introducing errors into these strings, testing error recovery and repair methods in parsers.

It might appear that this problem could be straightforwardly solved using the following naive algorithm. Starting with S, choose at random one of the productions of the form $S \rightarrow \alpha$ to generate a sentential form. Then repeat this successively expanding each leftmost non-terminal until all non-terminals have been replaced. This 'obvious' solution has two major problems. Firstly for complex grammars this process may either fail to halt or alternatively generate extremely long strings before it terminates. For example, given the grammar

$S \to SS|a$, the probability of there being zero S's left in the sentential form after any number of replacements is $\frac{1}{2}+\frac{1}{8}+\frac{2}{32}+\frac{5}{128}+\frac{14}{512}+\cdots$ which converges to approximately 0.873595. Secondly not all strings of the same length are equally likely to be generated. For example, given the grammar $S \to a|A; A \to b|c$, this will generate an a with probability $\frac{1}{2}$ while both b and c will only be generated with probability $\frac{1}{4}$. Furthermore it is clear that two different grammars that generate the same language could produce the same string with two distinct probabilities.

An alternative approach is to treat all strings of length n generated by the grammar equally so that each string is equally likely to be generated. Generating uniformly random strings was first considered by Arnold and Sleep [2] who presented an $\mathcal{O}(n)$ algorithm for generating balanced parenthesis strings which they required as skeletons of programs for their error repair scheme. The more general problem of generating uniformly random strings from a CFG was first considered by Hickey and Cohen [3] who presented two algorithms; a first that generated strings in time $\mathcal{O}(n^2(\log n)^2)$ and space $\mathcal{O}(n)$ and a second that works in time $\mathcal{O}(n)$ and space $\mathcal{O}(n^{|T|+1})$. More recently Mairson [5] improved on these time and space bounds, again presenting two algorithms; a first that generated strings in time $\mathcal{O}(n^2)$ and space $\mathcal{O}(n)$ and a second that works in time $\mathcal{O}(n)$ and space $\mathcal{O}(n^2)$. Both of these previous algorithms involve a pre-processing step that builds a data-structure which is used in a later generation phase. Although they analyse the storage requirements, the time complexity of this pre-processing phase is ignored in their analysis. Furthermore, Mairson's algorithm assumes the grammar is unambiguous and in Chomsky normal form without ϵ-productions.

In practice the pre-processing step is much more computationally expensive than the generation phase unless a large number of strings are generated. [1] In this paper we relax the requirement that the grammar is in Chomsky normal form (although we also require the absence of ϵ-productions) and present an algorithm whose pre-processing phase requires $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space. The subsequent string generation step generates a string in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.

It should be noted that the results of the pre-processing step are useful in themselves to perform a partial check that two distinct grammars that claim to generate the same language do indeed do so. This check involves comparing the number of strings of length 1,2,...,n generated by the start symbols of both grammars. If these differ then we can be sure the grammars do **not** generate the same language. Even though agreement in the counts does not

[1] Using the method presented in this paper for a grammar generating the Pascal language [4] approximately 700 Pascal programs, each of length 100 tokens can be generated in the time taken for the pre-processing phase.

guarantee the languages are the same, it can be a useful and strong check on similarity, especially if the counts are checked up to a large n.[2]

2 Notation

Let $G = (N, T, P, S)$ be a context free grammar with terminals Σ, non-terminals $N = \{N_1 \dots N_r\}$, start symbols $S \in N$ and productions

$$P = \{\pi_{ij} : N_i \to \alpha_{ij} | i = 1 \dots r, j = 1 \dots s_i\}$$

giving a total of $|P| = \sum_{i=1}^{r} s_i$ productions. The right hand sides of each production have the form

$$\alpha_{ij} = x_{ij1} \dots x_{ijt_{ij}}$$

giving a total of $|R| = \sum_{i=1}^{r} \sum_{j=1}^{s_i} t_{ij}$ symbols on right hand sides. The grammar G generates the language $L(G) \subseteq \Sigma^*$. The n-slice of $L(G)$ is the subset $L(G) \cap \Sigma^n$ containing all strings in $L(G)$ of length n.

In order to generate strings from the n-slice of $L(G)$ uniformly at random we require counts of the number of strings of length $k \leq n$ generated by the right hand side of each production α_{ij} and postfix parts of the production $x_{ijk} \dots x_{ijt_{ij}}$. To define these we shall use the notation $[c_i \mid i \leftarrow 1 \dots n]$ to represent the list $[c_1, c_2, \dots c_n]$, while $\|x\|_n$ gives the number of strings of length n generated by the symbol $x \in (N \cup T)$. Furthermore, we can extend this to $\|\gamma\|_n$ to act on $\gamma \in (N \cup T)^*$ using

$$\|x \bullet y\|_n = \sum_{l=1}^{n-1} \|x\|_l \times \|y\|_{n-l}$$

where $\bullet$ represents string concatenation. The limits of the sum are 1, $n-1$ rather than 0, n as we have disallowed any ϵ-productions in the grammar and so no symbol can generate the empty string.

[2] A suitable n is one that ensures that all productions of the grammar have been used. It is straightforward to find such an n during the pre-processing step.

3 Pre-processing

Let us define the following two functions:

$$f_i(n) = [\ \|\alpha_{ij}\|_n \mid j \leftarrow 1 \ldots s_j] \tag{1}$$

where $i = 1 \ldots r$, which is identical to the $N(A_i, n)$ function in [7] and

$$f'_{ijk}(n) = [\ \|x_{ijk}\|_l \times \|x_{ij(k+1)} \ldots x_{ijt_{ij}}\|_{n-l} \mid l \leftarrow 1 \ldots n - t_{ij} + k] \tag{2}$$

where $i = 1 \ldots r, j = 1 \ldots s_j, k = 1 \ldots t_{ij}$. The list $f_i(n)$ gives the number of strings of length n generated by each possible production with left hand side N_i. The list $f'_{ijk}(n)$ gives the number of strings of length n generated by the final symbols $x_{ijk} \ldots x_{ijt_{ij}}$ from the right hand side of the production π_{ij} for each of the possible ways in which the n symbols can be split between the first symbol x_{ijk} and the remaining symbols. Again the length of the list is $n - t_{ij} + k$ and not n because no symbol can generate the empty string and so all strings generated by the symbols $x_{ij(k+1)} \ldots x_{ijt_{ij}}$ will have length of at least $t_{ij} - k$.

The following algorithms can be used to calculate these count functions.

function $f_i(n)$
  /* *return a list giving the number of strings generated* */
  /* *for each production* $N_i \rightarrow \alpha_{ij}$ */
  return [sum $(f'_{ij1}(n)) \mid j \leftarrow 1 \ldots s_i$]
end

function sum(l)
  /* *given the list* $l = [l_1, l_2, \ldots l_m]$, *return* */
  /* *the sum of the elements* $\sum_{i=1}^{m} l_i$ */
end

function $f'_{ijk}(n)$
  /* *return the number of strings of length* n *generated* */
  /* *by the final symbols* $x_{ijk} \ldots x_{ijt_{ij}}$ *from the right* */
  /* *hand side of the production* π_{ij}, $N_i \rightarrow x_{ij1} \ldots x_{ijt_{ij}}$ */
  /* *for each of the possible ways in which the* n */
  /* *terminals can be split between the first symbol* x_{ijk} */
  /* *and the remaining symbols* */
  if $n = 0$ **then return** [ ]
  if $x_{ijk} \in T$ **then**
    if $k = t_{ij}$ **then**

```
        if n = 0 then return [1] else return [0]
      else
        return [sum (f'_{ij(k+1)}(n − 1))]
    if k = t_ij then
      return [sum (f_{x_ijk}(n))]
    else
      return [ sum (f_{x_ijk}(l)) × sum(f'_{ij(k+1)}(n − l)) | l ← 1 ... n − t_ij + k ]
  end
```

Expressing f and f' as functions suggests that evaluating $f'_{ijk}(n)$ will require $\mathcal{O}(n^{(t_{ij}-1)})$ calls to function f resulting in an exponential time complexity. However, in any practical implementation of this algorthm, results can be pre-computed and stored to avoid repeated re-evaluation. With the exception of chain rules, the calculation of $f_i(n)$ or $f'_{ijk}(n)$ will only result in calls to $f_i(m)$ and $f'_{ijk}(m)$ with $m < n$. This enables dynamic programming techniques to be used where the $m = 1$ values are first calculated and stored before calculating the $m = 2, \ldots$ values in turn. For chain rule productions of the form $N_i \rightarrow N_l$ care needs to be taken that the values involving N_l (namely, $f_l(m)$ and $f'_{l\ldots}(m)$) are evaluated before those involving N_i (namely, $f_i(m)$ and $f'_{i\ldots}(m)$). A more convenient approach is to regard f and f' as functions of two and four parameters respectively and then memoise [6] the functions over all these parameters. This approach automatically results in the order of non-terminals involved in chain rules being calculated in the correct order.

Either dynamic programming techniques or memoization significantly reduces the time complexity of the pre-processing phase with a corresponding space penalty. The space requirements for storing all values of $f_i(m)$ is a list of s_i elements for each of the r values of i and n values of m giving $n|P|$ elements. The space requirements for $f'_{ijk}(m)$ is a list of $n - t_{ij} + k$ elements for each of the r values of i, s_i values of j, t_{ij} values of k, and n values of m, giving $\mathcal{O}(n^2)$ elements in total. To calculate all possible values of $f_i(m)$ requires $n|P|$ calls to f' and calculating all possible values of $f'_{ijk}(m)$ requires $n \times (n - t_{ij} + k)$ calls to f and f' for each of the $|R|$ distinct triple of i, j, k values giving a total time complexity of $\mathcal{O}(n^2)$.

4 String Generation

To generate a string of length n uniformly at random from a non-terminal N_i, we consult the list $f_i(n)$. As this list gives the number of strings of length n generated by each possible production with left hand side N_i it determines the relative probability with which we should choose each production to select all strings uniformly at random. Hence we choose a production at random by

weighting each by the counts from this list. Formally, this is captured by the following function.

```
function g_i(n)
  /* generate a string of length n uniformly at random */
  /* from a non-terminal N_i                            */
  let r = choose f_i(n)
  in return g'_{ir1}(n)
end

function choose(l)
  /* given the list l = [l_1, l_2, ... l_m], return an */
  /* index i between 1 and m at random with            */
  /* probability l_i / sum_{j=1}^{m} l_j                */
end
```

Having selected a production $N_i \rightarrow x_{ij1} \ldots x_{ijt_{ij}}$ to generate a string of length n we then need to decide how to split n amoung the t_{ij} symbols on the right hand side of this production. This is achieved by first deciding on the split between first symbol x_{ij1} and the remainder and then repeating this recursively. To decide this split we can use the list returned by $f'_{ijk}(n)$ which gives the number of strings of length n generated by the final symbols $x_{ijk} \ldots x_{ijt_{ij}}$ from the right hand side of the production π_{ij} for each of the possible ways in which the n symbols can be split between the first symbol x_{ijk} and the remaining symbols. Hence we choose a split at random by weighting each split by the counts from this list.

```
function g'_{ijk}(n)
  /* generate a string of length n uniformly at random */
  /* from among all strings derivable from the symbols */
  /* x_{ijk} ... x_{ijt_ij} taken from the right hand side of the */
  /* production N_i -> alpha_ij                         */
  if x_{ijk} ∈ T then
    if k = t_ij then
      return x_{ijk}
    else
      return x_{ijk} • g'_{ij(k+1)}(n - 1)
  if k = t_ij then
    return g_{x_{ijk}}(n)
  else
    let l = choose f'_{iik}(n)
    in return g_{x_{ijk}}(l) • g'_{ij(k+1)}(n - l)
end
```

Generating a string of length n from the start symbol requires repeatedly choosing a production (taking constant time) and then selecting a split for each possible non-terminal on the right hand side. There can be no more than n splits required to generate the string so the total time complexity is $\mathcal{O}(n)$. The space requirement is the need to store the sentential form from the leftmost non-terminal which requires $\mathcal{O}(n)$ space in the worst case.

5 Concluding Remarks

The form of equation 2 has a structure remarkably similar to the convolutions

$$z_n = \sum_{l=0}^{n-1} x_l y_{n-l} \tag{3}$$

calculable in $\mathcal{O}(nlogn)$ steps using Fast Fourier Transform (FFT) techniques[1]. The similarity can be seen by replacing z_n by $f'_{ijk}(n)$, x_l by $f_{x_{ijk}}(l)$ and y_{n-l} by $f'_{ij(k+1)}(n-l)$. The FFT method is to calculate vectors x and y in $\mathcal{O}(nlogn)$ steps, perform the inner product in $\mathcal{O}(n)$ steps and finally calculate the vector z using the inverse FFT also in $\mathcal{O}(nlogn)$ steps.

It does not seem that this technique can be applied here as the x, y and z in equation 3 are independent, while in our situation any grammar with recursive production rules will mean they are mutually dependent. However the FFT method does suggest the generation step might be able to be reduced to $\mathcal{O}(nlogn)$ complexity.

Finally it should be noted that if the grammar is ambiguous, strings with more than one possible derivation will be counted multiple times by our algorithm and will hence be generated with a correspondingly increased probability.

Acknowledgement

The author would like to thank Tadao Takaoka for helpful comments and discussions.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] D. B. Arnold and M. R. Sleep. Uniform random generation of balanced parenthesis strings. *ACM Trans. Prog. Lang.*, 2(1):122–128, 1980.

[3] Timothy Hickey and Jacques Cohen. Uniform random generation of strings in a context-free language. *SIAM J Comput.*, 12(4):645–655, November 1983.

[4] Kathleen Jensen and Niklaus Wirth. *Pascal : User Manual and Report.* Springer-Verlag, New York, 1975.

[5] Harry G. Mairson. Generating words in a context-free language uniformly at random. *Information Processing Letters*, 49(2):95–99, 28 January 1994.

[6] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[7] Tadao Takaoka. A definition of measures over language space. *Journal of Computer and System Sciences*, 17(3):376–387, December 1978.