Inferring Lindenmayer Systems

Eric Ung*

University of Minnesota
ungxx011@umn.edu

December 30, 2018

Abstract

Lindenmayer systems are formal grammars that rewrite sentences using a set of rules. Investigation on the use of Lindenmayer systems and the possibility of inferring it so that it can predict subsequent generations. The practicality of this is that on occasions where the rules of the pattern don't exist then there exists an algorithm that could possibly help predict the next derivations. In addition to this, an example to the question of what to use to model abstract concepts to infer completely or learn different structures is shown.

I. Introduction

Lindenmayer systems are structures that were originally developed by Aristid Lindenmayer to describe how plants grow. They are simple systems that can be used to model generative concepts. Investigation on learning them will be constructed to show they are easy to understand and able create more complicated ideas. The first part of this paper is to define the structures to be used which are context-free grammars and, more specifically, deterministic and context-free Lindenmayer, D0L, systems.

II. Definition of Context-Free Grammmar

In order to understand Lindenmayer systems and the problem, context-free grammar will need to be defined. A context-free grammar is a system that describes a language made up of substitution rules, variables, terminals, and a start variable defined as follows:

$$CFG = (V, \Sigma, R, S)$$

V is the variables, Σ is the terminal, R is the set rules, and $S \in V$ is the start variable. An example of this is as follows.

$$G = (S, a, b, R, S)$$
$$R = \{S \rightarrow abS|\epsilon\}$$

*

The above example starts with S, then G proceeds to keep generating the sequence of string ab or stops with S being ϵ like the following.

$$S \Longrightarrow abS \Longrightarrow ababS \Longrightarrow abab$$

This example shows that S's derivation becomes a concrete value, terminal, until S decides to become an empty value, ϵ.

III. Definition of D0L system

Unlike formal grammars where the variables and terminals are separate, Lindenmayer systems only have variables. The simplest Lindenmayer system that can be described is a D0L system, deterministic and context-free, defined as:

$$L = (V, \omega, P)$$

V are the variables, ω is the starting variable, and P is the set of production rules. An example is as follows:

$$L = (\{a, b\}, a, P)$$
$$P = \{a \rightarrow ab, b \rightarrow a\}$$

The derivation sequence this grammar gives is:

$$a \Longrightarrow ab \Longrightarrow aba \Longrightarrow abaab \Longrightarrow abaababa \tag{1}$$

The variables in this case is both the variable and terminal found in context-free grammars. The definition will now allow the proposal of learning the next sequence when the production rule is unknown.

If a Lindenmayer is composed of previous derivations, it is locally catenative. An example of this is the Fibonacci words, as an example (1). It takes the last two previous derivations and composes it into the next word in the sequence. More formally, this can be described as:

$$w_{\mathrm{n}} = w_{\mathrm{n\text{-}i_1}}, ..., w_{\mathrm{n\text{-}i_k}} where\, n \geq max\{i_1, ..., i_{\mathrm{k}}\}$$

A locally catenative D0L system is described by the number of previous words in the derivation sequence the word contains called a cut. For instance if a word w_n is composed of previous derivations, ie. $w_n = w_{n-1}w_{n-2}w_{n-3}$ then this would be a system that has a cut of three.

IV. Problem Statement

The problem of this paper is to find the next word given the minimal amount of words given a locally catenative system. The production rules are the unknown in this situation.

Given a sampling sequence V = {w_1,...,w_n}, w_{n+1} can be predicted where w_i is an arbitrary word in the derivation sequence, $1 \leq i \leq n$, $n \geq$ |variables in grammar|, and that the w_i derives w_{i+1}. The solution to this problem will be derived in the next sections. The production rules are such that it doesn't lead to an empty value or itself. The solution is split up into three parts. The first part is to check if the Lindenmayer is composed of previous derivations. The procedure for this is called *hasAllDerivations*. The second part is to actually find the order of the sequence the Lindenmayer system is which will be called $findSequence$. The third and last part is to generate the new sequence given that *hasAllDerivations* confirms that the sampling sequence has some relation in the derivations and $findSequence$ is executed to find the sequence order. This is called *generateNext*. It is expected that the input is not null and that the number of parameter, *derivations*, is more than or equal to the number of terminals in the system being inferred.

V. Part I of Solution

The first part of the solution is to show that previous Lindenmayer systems compose the current derivation. The algorithm needs to have a series of previous derivations that are consecutive in addition to the final derivative.

Algorithm 1 hasAllDerivatives

```
1:  procedure hasAllDerivatives(derivations)
2:      n ← derivations.Length
3:      tempString ← derivations[n]
4:      reverseDerivations ← derivations[1...n-1]
5:      returnValue = false
6:      for all entry in derivations do:
7:          if tempString.Contains(entry) then:
8:              tempString ← tempString.Replace(entry, "")
9:      if  then(temp == string.Empty)
10:         returnValue = true
11:     return returnValue
```

The algorithm will be shown to terminate. Lines 2 to 4 sets the initial variables to be used. The need to reverse *derivations* in step 3 comes from the fact that the first entries in *derivations* might be contained in later entries of it. In essence, taking the larger entries of *derivations* will allow the algorithm to remove sub-strings of *tempString* without giving a string that is incoherent. The next part is the for loop in lines 5 to 7. It will go through all the entries in derivations and if it contains the entry in question, it will replace *tempString* with the empty string. Once the for loop is done, the next part is lines 8 to 9 which determines if the derivations make up the string in question, *derivative*. The returned value is true if *tempString* is found to be empty implying that *derivative* contains some or all of *derivations* and false if otherwise. This explanation shows that the algorithm will terminate.

The next step is to show the running time. Lines 2 to 4 are initiation steps taking O(1) time each. Lines 5 to 7 is a for loop iterating through each *entry* of *derivations* in O(derivations) time. Line 6 is a run-time of O(|*tempString*|). Line 7 is also the same with run-time of O(|*tempString*|). The total time is then O(derivations*max(|derivations.Length|).

VI. Part II of Solution

The second part of the solution once it is verified that previous derivations compose the final derivation of the sequence is to find the sequence it composes. This means what order w_n composes.

Algorithm 2 findSequence

```
1:  procedure findSequence(derivations)
2:      n ← derivations.Length
3:      tempEmpty ← derivations[n]
4:      tempReplace ← derivations[n]
5:      index = 1
6:      for all element in derivations[1...n − 1].Reverse() do:
7:          if tempEmpty.Contains(element) then:
8:              tempEmpty ← tempEmpty.Replace(element, "")
9:              tempReplace ← tempReplace.Replace(element, index)
10:         index++
11:     prev = −1
12:     sequence = newArray(n − 1)
13:     place = 1
14:     for all element in tempReplace do:
15:         if prev! = element then
16:             prev = element
17:             sequence[place] = element
18:             place++
19:     return sequence
```

To show that it terminates, the first lines, 2 to 5, are initialization of the variables. The for loop of lines 6 to 10 iterate through each element of derivations minus the last element. In line 7, the element Contains operates as a boolean function and checks if the substring *element* is contained in *tempEmpty*. If it isn't, it stops right there, otherwise, it calls Replace twice which terminates once it finds the words. Lines 11 to 13 initialize variables for the upcoming for loop. Lines 14 to 18, *tempReplace* is iterated through all of it's characters. Each character is checked to see if it is the first one in the sequence and if it is, it gets added to the return value, *sequence* and the place holder, *place*, gets updated.

To show the time complexity of this algorithm, start with lines 2 to 5. This takes constant time. Iterating through lines 6 to 10, the for loop goes through each element of derivations subtracted by one element. In line 7, the Contains function goes through to compare the *tempEmpty* variable with each *element* from *derivations*$[1...n-1]$. At most, this takes O(|*tempEmpty*|). Lines 8 and 9 have both Replace as a function taking O(|*tempEmty*|) time. Line 10 takes O(1) time. In total, lines 6 to 10 take O(max|*element*|*(derivations.Length - 1)). The next lines, 11 to 13, take O(1) time. Next, the for loop of lines 14 to 18 iterate through each character *tempReplace*. The operations in lines 15 to 18 take O(1) time. This means that lines 14 to 18 runs in time O(|*tempReplace*|). In total, the complexity of this function gives O(max|*element*|*(derivations.Length - 1) + |*tempReplace*|) = O(max|*element*|*(derivations.Length - 1)) = O(m*n) such that m = max|*element*| and n = derivations.Length.

VII. Part III of Solution

The final part of the solution is the combination of the previous two algorithms to check and find the composition of the last derivation in the derivation list to create the predicted value.

Algorithm 3 generateNext

```
1: procedure generateNext(derivations)
2:     u_{n+1} = string.Empty
3:     if hasAllDerivations(derivations) == true then
4:         return u_{n+1}
5:     sequence = findSequence(derivations)
6:     for all element in sequence do:
7:         u_{n+1} += derivations[element]
8:     return u_{n+1}
```

To show that the algorithm terminates, start with line 2 which terminates. Line 3 calls hasAllDerivations which is proven to terminate and returns an empty string u_{n+1}. Line 5 calls findSequence which is also proven to terminate and returns an ordered array of numbers representing the derivations. The for loop in lines 6 to 7 iterates through each element once and ends. Line 8 returns u_{n+1} whether it is an empty string or not.

To analyze the time complexity of this algorithm, start with line 2 which takes O(1) time. The next step is step 3 which calls hasAllDerivations taking O(m*n) time. Line 4 returns u_{n+1} if it is correct. Line 5 is findSequence which takes O(m*n) time. The next steps, lines 6 to 7, takes O(|sequence|) times as it is a for loop. The ending analysis gives a time complexity of O(m*n) time.

VIII. Post Analysis

The first issue is what happens when there is a missing derivation k such that $1 \leq k \leq n$. Is there a way to predict what the missing value is?

$$u_{i_1} \Longrightarrow ... \Longrightarrow u_{i_{k-1}} \Longrightarrow u_{i_{k+1}} \Longrightarrow ... \Longrightarrow u_{i_n}$$

In addition to this, there is an interesting feature in some of the structures of D0L systems. When looking at changing the starting variable, it can be seen that some systems will be a permutation of each other keeping the locally catenative property while others will not contain the property, ie these three examples below are locally catenative.

$$G = (\{a, b, c, d\}, a, \{a \to bd, b \to a, c \to b, d \to bc\})$$
$$G = (\{a, b, c, d\}, b, \{a \to bd, b \to a, c \to b, d \to bc\})$$
$$G = (\{a, b, c, d\}, c, \{a \to bd, b \to a, c \to b, d \to bc\})$$

Whereas this example starts with the variable, d, but the following is not locally catenative.

$$G = (\{a, b, c, d\}, d, \{a \to bd, b \to a, c \to b, d \to bc\})$$

The last question is if there is there an algorithm that will find the rules. This means if there is a way to generate rules backwards then it can be found that each variable corresponds to a production rule.

IX. Conclusion

It is shown that w_{i+1} can be predicted with a derivations V = {w_1,...,w_n} such that $1 \leq i \leq n$, $n \geq |$variables in grammar$|$ and that the production rules do not derive the empty value or itself. This implies that locally catenative D0L systems can be inferred when there is more than or equal sequential derivations than the amount of variables.

References

[Michael Sipser, 2006] Sipser, M. (2006). Introduction to the Theory of Computation.

[Grzegorz Rozenberg and Arto Salomaa, 1980] Rozenberg,G. and Salomaa, A. (1980). The Mathematical Theory of L Systems *Academic Pres*, 10–42.

[Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, 2009] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). Introduction To Algorithms.