Algorithm Outline for Tree Context Matching

Alex Carney

May 2022

1 Introduction

The *Turtle Time* application generates L systems using an axiom and a set of productions. Each iteration, the set of productions acts on the axiom, producing an output string that becomes the axiom for the next iteration.

Our contract requires the implementation of OL-deterministic L systems, OL-stochastic L systems, IL-deterministic systems, and IL-stochastic systems.

Additionally, the program generates tree-deterministic L systems, and tree-stochastic L systems.

Context dependent tree L systems were part of our "extra proficiencies we may or may not deliver" section. We were unable to implement a working algorithm for this class of systems. However, this paper proposes an solution for the implementation of such an algorithm.

In section 1, we discuss the goal of context dependent tree systems, and the nature that sets them apart from all the other class of systems we successfully implemented.

In section 2, we discuss the details of an algorithm that would solve this problem.

2 Context Dependent Strings

For non-tree systems, implementation of context sensitivity is relatively straight forward. When a production is applied to an axiom, the 'left context' and 'right context' are strings that come before or after the letter that is being acted upon. Since the length of each context is known to the algorithm (since it is defined in the production itself), the operation simply has to trace backwards/forwards n characters in the current L string, and execute a string .equals() to the context defined in the production.

As an example, consider the following axiom word, and context sensitive production:

$$\begin{aligned} \omega &: \text{AABBABCBABCAB} \\ p_1 &: \text{AB} < \text{C} > \text{BAB} \rightarrow \text{A} \end{aligned} \tag{1}$$

Our L-string algorithm would loop through the word ω until C is reached. At this point, the context-sensitive production would be fired. If the context matches, the production is applied.

First, the left-context is analyzed. In the definition of p_1, the left context is AB, with a length of 2. This means that our procedure *knows* to check the substring containing the previous two characters in ω, and nothing else.

The previous 2 letters are extracted, yielding $AB \in \omega$. This is compared using string equality to the left context $AB \in p_1$, which yields true. Then, the right context is analyzed. The right context is BAB, with a length of 3. Our procedure knows to check the substring containing the next three characters in ω, and nothing else.

The next 3 letters are extracted, yielding $BAB \in \omega$. This is compared to the right context $BAB \in p_1$, which yields true. Since both the left and right contexts yield true, the procedure is applied, and C is replaced with $A \in p_1$.

Here are the key concepts to take away from this approach:

1. The *length* of the left context is an attribute that can be exploited to extract the pre-context from ω to be compared

2. A .equals() method comparing the data type of contexts (in this case, strings) is required

As we will see in section 2, our proposed algorithm uses the exact same concepts as described above. The only difference is an extreme overhead in code implementation, due to our data type no longer being strings.

3 Problem Setup

Tree systems are represented using bracketed string notation. As stated in the algorithmic beauty of plants: *The introduction of context to bracketed L-systems is more difficult than in L-systems without brackets, because the bracketed string representation of axial trees does not preserve segment neighborhood. Consequently, the context matching procedure may need to skip over symbols representing branches of branch portions.*

We will use this author's example in our explanation of our algorithm.

A context sensitive tree production has three components: *pathl* takes the role of the pre-context, *edge S* is the strict predecessor, and *axial tree r* takes the role of the right context. An immediate note is that the pre and post contexts are asymmetric, adding to the complexity of the problem.

The figure below demonstrates a production in frame a, and its detection in the tree in frame b. Our algorithm will detect whether or not this context is present in a given tree.

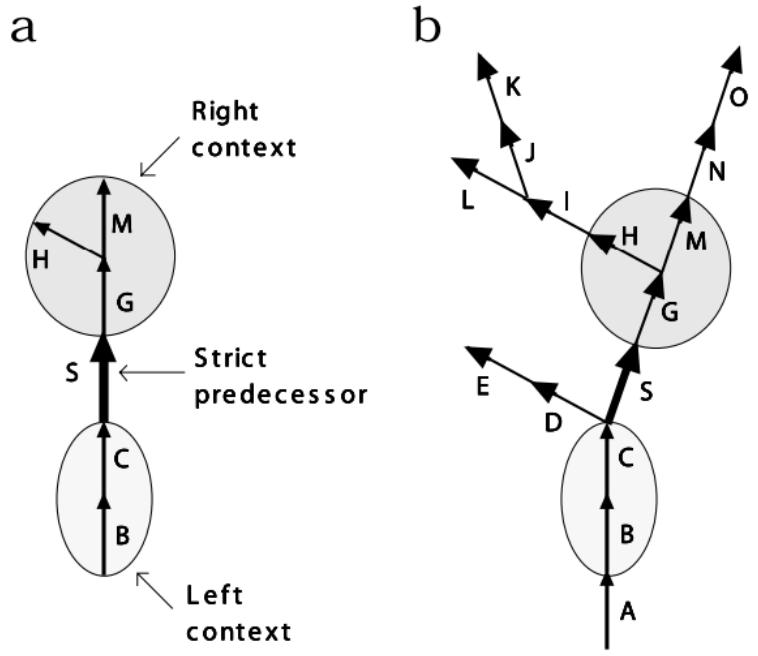

Figure 1: Frame a depicts the production itself. Frame b depicts detection of the proper context inside of a given tree.

According to Springer, frame a above represents the production written as $p = BC < S > G[H]M$, and the full tree in frame b is represented by the string:

$$\omega = \text{ABC[DE][SG[HI[JK]L]MNO]} \tag{2}$$

It is much more difficult to detect the presence of p in ω when looking at their string forms. The algorithm would have to skip over $[DE]$ in pursuit of the left context (which is actually relatively straight forward, since the left context is a 'path'), but would have to skip over $I[JK]L$ in search of the right context (considerably more difficult).

It is likely possible to devise an algorithm that reads ω in string form and can successfully detect context required for p, but our algorithm will take a different approach.

4 A Proposed Algorithm

Our algorithm should seem strikingly similar to our algorithm for string context matching.

The main difference is in our data structure. For strings, the data structure is simply strings. Backtracking over strings is completely straight forward, as is

the implementation of .equals(). For trees, we have a completely different data structure – Directed acyclic graphs (DAGs). In fact, our exact data structure is actually a subset of DAGs, because we are able to impose an extra condition (each node has only one 'parent' node).

I couldn't find a name for this subset of DAGs, meaning searching/writing algorithms for equality and backtracking would have to be done from scratch. I'm going to refer to this data structure as DSPAGs (directed single parent acyclic graphs) for the rest of this paper.

A DSPAG is a data structure made up of nodes and edges. Although DAGs are usually represented using an adjacency list/matrix, we will take a different aproach, more akin to the representation of binary trees.

A Graph Node class would have the following members:

Listing 1: GraphNode class

```
class GraphNode:
data: Char
parent_node: GraphNode
child_nodes: List[GraphNode]
```

Therefore, a DSPAG would simply be a root GraphNode that stores its children inside of its list of child nodes. This is similar to a linked list or a binary tree, and may not be the most efficient storage of our data structure, but will come in handy later.

Step 1 of the algorithm is converting the bracket string notation into a DSPAGs. Bracket string notation is a common way to represent binary trees, and we were able to get a bracket string to binary tree converter working.

Step 2 is converting the left and right contexts into DSPAGs as well. This is imperative for later stages of the algorithm, where we are comparing subgraphs of the entire tree to these contexts for equality.

At this point, our program would have access to the string representation of the DSPAG (the word ω itself), *and* an object in memory representing a collection of GraphNodes, with their connections defined through lists of parents and child nodes.

Next, our L-tree algorithm would loop through the word ω until the strict predecessor is found. At the same time, the algorithm would also traverse through the DSPAG data structure, using a depth-first-style (DFS) form of traversal (because that is identical to how the string is written). The algorithm would ensure that the pointer in ω and the pointer inside the DSPAG are synchronized.

Since the strict predecessor is a single edge (S), we can use character matching for this. At this point, the context sensitive production would be fired.

First, the left context is analyzed. In the definition of p, the left context is a path BC. Since BC is two characters, we know that the left context will be a *path* with size 2.

Our algorithm knows where we are in the DPSAG. Since a path only consists of parent nodes, our algorithm would extract a sub-graph from the previous 2

parent nodes, starting from S.

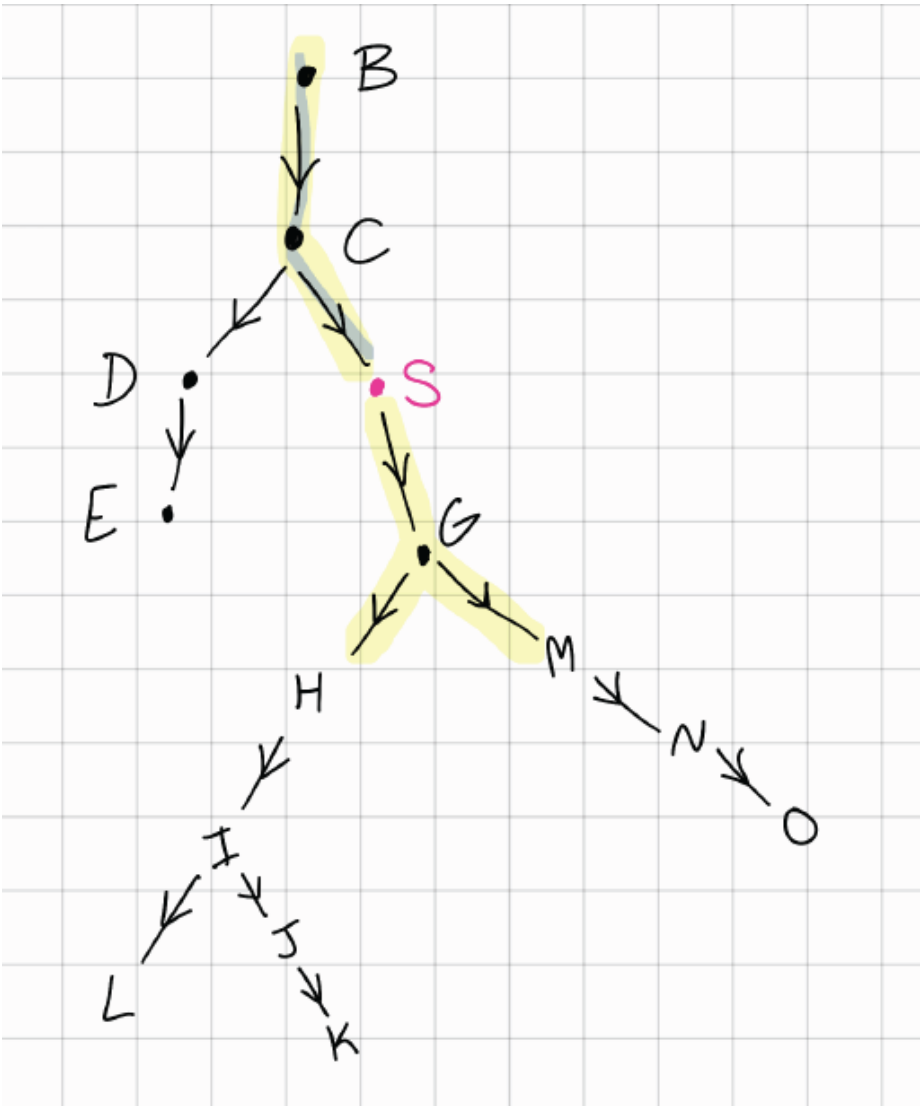

Figure 2: Our algorithm backtracks through only the parent nodes 2 times (the size of BC)

A subgraph consisting of BC is extracted, and compared to our DSPAG representing BC which we obtained from Step 2. A .equals() method for DSPAGs would have to be implemented for this to work. A .equals() implementation would draw inspiration from the same method for binary trees, but have added complexity due to the unknown amount of child nodes.

Listing 2: Graph .equals()

```
import numpy as np
def equals(a, b):

# Both Empty
if a is None and b is None:
    return True

# Both non empty, compare
return a.data == b.data
and
np.array_equals(a.child_nodes, b.child_nodes)
and
equals(a, b) for a in a.child_nodes for b in b.child_nodes

# One empty, one not
return False
```

(Above code inspired by this article)

Since the comparison between DSPAGs represented by BC and BC would return true, the right context would be tested next.

The right context is not a path, it is an axial tree. This means we cannot backtrack in the easy sense (go up 2 parent nodes to extract a subgraph). Instead, we need to exploit the 'backtracking' strategy in computer science.

According to wikipedia, backtracking is a strategy *that incrementally builds candidates to the solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution.*

Therefore, we employ the backtracking technique starting at S to extract **all possible subgraphs** of size 3 (since $G[H]M$ is of size 3)

The backtracking algorithm would perform a DFS search starting from S going "forwards" (meaning only using child nodes in its solution). Every subgraph of size 3 would be added to a list of potential candidates, and immediately compared (using .equals() again!) to the correct right context.

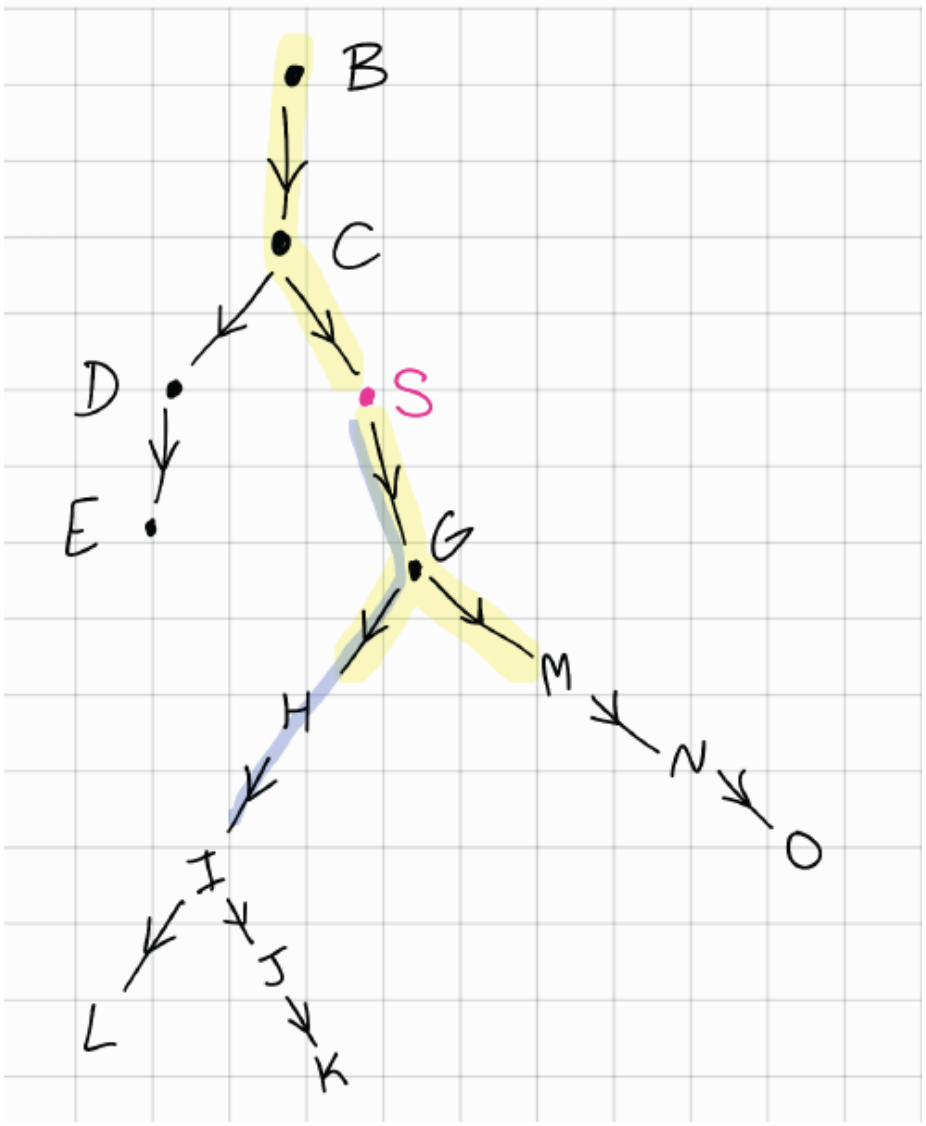

Figure 3: The first subgraph of size 3 is extracted and thrown away, it would fail an equality check to the graph represented by $G[H]M$

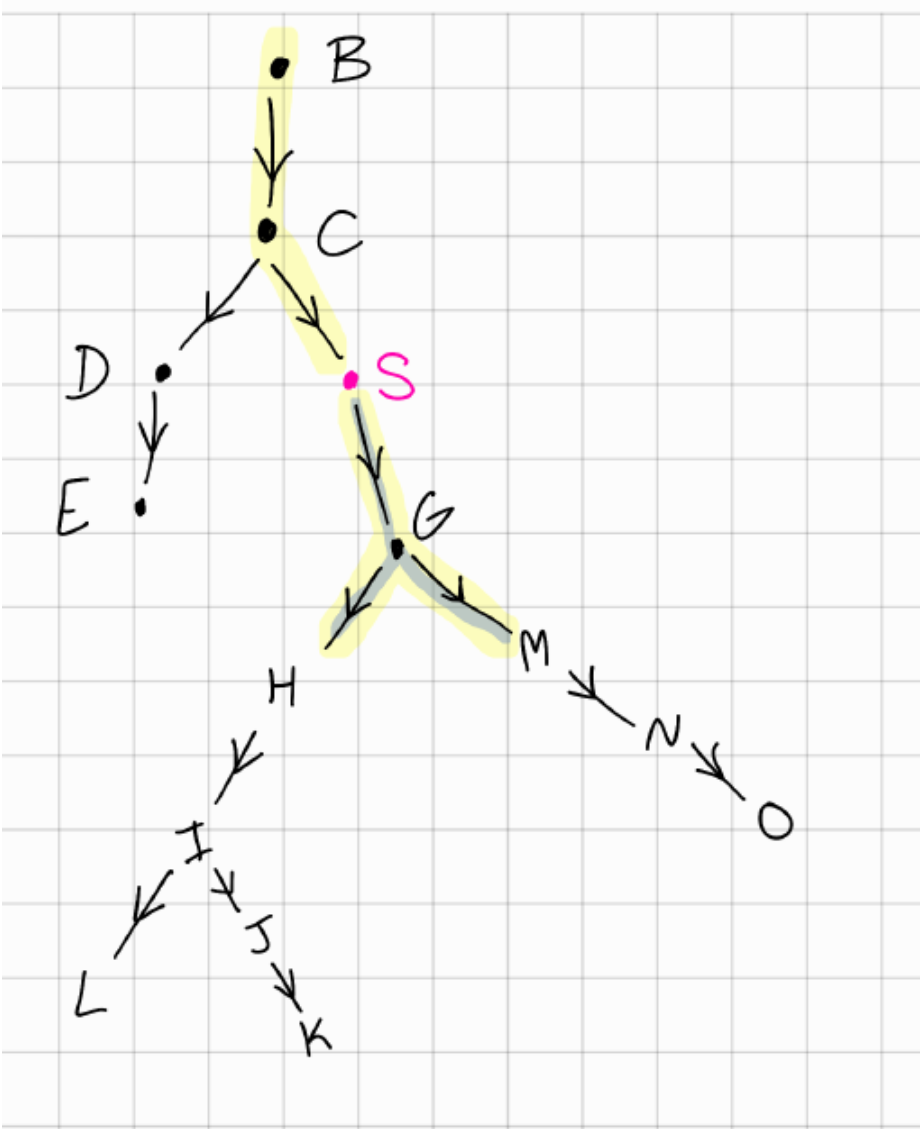

Figure 4: The second subgraph of size 3 is extracted and accepted, it would pass an equality check to the graph represented by $G[H]M$

At this point, our algorithm would successfully deem the context requirements met, and the production would be applied, replacing *edge* S by any successor (with simple string replacement. The graph does not have to be updated in this iteration, instead, the word is updated, and during the next iteration the entire string is converted into a DSPAG again.)