D^3 as a 2-MCFL

Abstract. We discuss the open problem of parsing the Dyck language of 3 symbols, D^3, using a 2-Multiple Context-Free Grammar. We tackle this problem by implementing a number of novel techniques and present the associated software packages we developed.

Keywords: Dyck Language; multiple context free grammars (MCFG)

1 Introduction

Our goal with this paper is the analysis of the 3-dimensional Dyck Language, D^3, under the scope of a 2-multiple context-free language, 2-MCFL. For brevity's sake, this chapter only serves as a brief introductory guide towards relevant papers, where the interested reader will find definitions, properties and various correspondences of the problem.

D^3 is defined over a lexigraphically ordered alphabet (a, b, c) as the generalized language of well-balanced parentheses[**?**]. We attempt to model D^3 via a 2-MCFG; the class of multiple context free grammars that operate on pairs of strings[**?**].

There are a number of interesting correspondences to D^3. Firstly, a word of D^3 can be presented as a *standard Young Tableau*, which is a rectangular table with strictly ascending rows/columns containing as entries the numbers $\{1, 2, \ldots, n\}$ where n the total number of symbols in the word. The Young Tableau can be obtained by placing (in order) each character's index to the row corresponding to its lexicographical ordering (e.g. an a is placed on the first row)[**?**].

Another correspondence exists between D^3 and combinatorial *Spider Webs*, a special category of directed planar graphs embedded on a disk[**?**]. Spider Webs can be obtained through the application of a set of rules, known as the *Growth Algorithm*, which operates on pairs of neighbouring nodes, collapsing them into a singular intermediate node, transforming them into a new pair or eliminating them altogether. At termination, this process produces a well-formed Spider Web, which, in the context of D^3, can be interpreted as a visual representation of parsing a word.

A bijection links Young Tableaux with Spider Webs. Specifically, the *Jeu-de-taquin* algorithm can be applied on a Young Tableau, which transforms it to another one through an act called *promotion*. Subsequent promotions will eventually result in the initial tableau. Promotion thus defines an equivalence class, which we call an *orbit*[**?**].

2 Modeling Techniques

We now present a number of novel techniques that we developed as an attempt to solve the problem at hand, incrementally moving towards more complex and abstract grammars. For the purpose of experimentation we have implemented these techniques, based on a software library for parsing MCFGs[**?**].

2.1 Triple Insertion

To set things off, we start with the grammar of *triple insertion*. This grammar operates on non-terminals $\mathsf{W}(x, y)$, producing a $\mathsf{W}(x', y')$ with an additional triplet *a, b, c* that respects the partial orders $x < y$ and $a < b < c$. The end-word is produced through the concatenation of (x, y).

$$\mathsf{S}(xy) \leftarrow \mathsf{W}(x,y). \tag{1}$$
$$\mathsf{W}(\epsilon, xy\textbf{abc}) \leftarrow \mathsf{W}(x,y). \tag{2}$$
$$\dots$$
$$\mathsf{W}(\textbf{abc}xy, \epsilon) \leftarrow \mathsf{W}(x,y). \tag{61}$$
$$\mathsf{W}(\epsilon, \textbf{abc}). \tag{62}$$
$$\dots$$
$$\mathsf{W}(\textbf{abc}, \epsilon). \tag{65}$$

Fig. 1. Grammar of triple insertions

$$\mathsf{C}^-(\epsilon, xyzw) \leftarrow \mathsf{A}^+(x,y),\ \mathsf{B}^+(z,w).$$
$$\mathsf{C}^-(x, xyzw) \leftarrow \mathsf{A}^+(x,y),\ \mathsf{B}^+(z,w).$$
$$\mathsf{C}^-(xy, zw) \leftarrow \mathsf{A}^+(x,y),\ \mathsf{B}^+(z,w).$$
$$\mathsf{C}^-(xyz, w) \leftarrow \mathsf{A}^+(x,y),\ \mathsf{B}^+(z,w).$$
$$\mathsf{C}^-(xyzw, \epsilon) \leftarrow \mathsf{A}^+(x,y),\ \mathsf{B}^+(z,w).$$

Fig. 2. Interaction between A^+ and B^+

Despite being conceptually simple, this grammar consists of a large number of rules. Its expressivity is also limited; the prominent weak point is its inability to manage the effect of *straddling*, namely the generation of words whose substituents display complex interleaving patterns.

2.2 Adding States: Meta-Grammars

To counteract the limited performance of the above grammar, we introduce more states to describe incomplete words; that is, words that either have an extra symbol or miss one. The former are positive states, whereas the latter are negative. The addition of these extra states would allow for more intricate interactions. For instance, a state A^+ could interact with a B^+ to produce a C^- in a multitude of different ways. Interestingly, there is a direct correspondence between these states and the nodes of Petersen's growth algorithm.

These new emerging interactions vastly increase the amount of rules that need to be considered. Moreover, for each pair interaction, different constraints pertaining to the order of the permuted concatenated strings need to be preserved. This necessitates a more abstract view of the grammar as whole. To that end, we define $\mathcal{O}$ as the *meta-rule* which, given a rule format, a set of partial orders (over the tuple indices of its premises and/or newly added terminal symbols), and the MCFG dimensionality, automatically generates all the order-respecting permutations. We thereby reduce the manual labour required to transcribe and debug individual rules.

$$\mathcal{O}_2[\![\mathsf{C}^- \leftarrow \mathsf{A}^+, \mathsf{B}^+ \mid \{t_{11} < t_{12} < t_{21} < t_{22}\}]\!].$$

As an example, we give the above meta-rule which demonstrates the interaction between A$^+$ and B$^+$ to produce a new C$^-$ state, as depicted in Fig.2.

This additional abstraction, however, is not enough on its own. We still needed the ability to enforce additional constraints related not to the order, but rather the positions of concatenated strings. To make this clear, let us present the unique case of B^-. Unlike other negative states, $B^-(t_{11},\ t_{12})$ has no direct closure unless we actually know it to contain its extra a in the left-side string t_{11} and its extra c in t_{12} (we refer to this distinction between a position-blind and a position-aware state as *refinement*). In that case alone are we allowed to insert the missing b (e.g. as obtained from a $B^+(t_{21},\ t_{22})$ between t_{11} and t_{12}. But in order to automatically generate B^- states for which this additional condition holds, we need to keep note of the origin of the extra a and c, and filter out the rules generated by $\mathcal{O}$ accordingly. We thus define $\mathcal{C}$ as a stricter version of $\mathcal{O}$, which constraints the rules according to the positions the premise tuples may occupy (after concatenation) in the conclusion's tuple.

$$\mathcal{C}_2[\![\mathsf{B}^- \leftarrow \mathsf{A}^+, \mathsf{C}^+ \mid \{t^l_{11} < t^l_{12},\ t^r_{21} < t^r_{22}\}]\!].$$

In the above notation, we illustrate that B$^-$ may be constructed from A$^+(t_{11},\ t_{12})$ and C$^+(t_{21},\ t_{22})$ only as long as the partial orders $(t_{11},\ t_{12})$ and $(t_{21},\ t_{22})$ are independently satisfied and t_{11}, t_{12} are both placed on the left element of the new state's tuples, while t_{21} and t_{22} on the right. This of course only allows for C$^-$ to consist of the tuple $(t_{11}t_{12},\ t_{21}t_{22})$.

The above abstractions significantly accelarated the process of creating and testing grammars, as they allowed us to simply describe rules instead of explicitly defining them. We thus moved from the lower level of grammars to the higher level of *meta-grammars*, or grammars of grammars. This strikes a resemblance to the notion of *declarative programming*, where one only cares about what needs to be computed, rather than how.

2.3 Refining States: Rule Inference

Perhaps expectedly, the improved performance of our new approach again proved insufficient to completely parse D^3. New problems started arising when trying to solve longer words; even though they were sound, our rules failed to make full use of the expressivity that a 2-MCFG allows. We soon realized that we have over-constrained our rules by making them comply to the worst-case scenario. Following the example of Fig. 2, if we had kept memory of the position of a in a *refined* version of $A^{+}(x, y)$, we could allow new rules to exist, such as

$$\mathsf{C}^{-}(xz, wy) \leftarrow \mathsf{A}^{+}_{left}(x, y),\ \mathsf{B}^{+}(z, w).$$

in the case of the extra a residing in x. Splitting each state into multiple different, refined states that now provide knowledge of their extra characters' positions, we could model vastly more interactions.

At this point, the abstraction offered by our meta-grammars did not cover our needs anymore. The same difficulty that we had encountered before was prominent once more, except now at an even higher level. Transcribing meta-rules and proofreading them was time consuming and error-prone, effectively hindering our progress. In order to test a new hypothesis pertaining to the refined states we would again have to go through a lot of arid work.

As a solution to the aforementioned limitations, we designed a system that can automatically create a full-blown m-MCFG given only the states it consists of. To accomplish this, we assign each state a unique *descriptor* that specifies (up to an extent) the content of its tuple's elements. Aligning these descriptors with the tuple, we can then infer the descriptor of the resulting tuple of every possible state interaction. For the subset of those interactions whose resulting descriptor is matched with a state, we can now automatically infer the rule.

Formally, the system is initialized with a map $\mathcal{D}$, whose domain, $dom(\mathcal{D})$, is a set of *state identifiers* and its codomain, $codom(\mathcal{D})$, is the set of their corresponding *state descriptors*.

Note that the above algorithm can easily be generalized to be applicable to the combination of an arbitrary number of states, but doing so would render it computationally infeasible. By using *ARIS*, one needs only specify a grammar's states and its descriptors, thus eliminating the need to explicitly define rules or even meta-rules.

3 Implementation

Enabled by the machinery described in the previous chapter we performed a multitude of tests using different grammars. In this chapter we describe a few of these grammars that we deem as milestones of our progression, in ascending order of complexity. We also present some statistics to allow for a direct comparison to be made between them. For the purposes of this project we use a version of MCFParser [**?**] as our backend, which we adapted to our needs[1].

[1] The original can be found at https://github.com/heatherleaf/MCFParser.py

Algorithm 1 ARIS: Automatic Rule Inference System

```
procedure aris($\mathcal{D}$)
    for $X \in dom(\mathcal{D})$ do
        $(d_1, \ldots, d_n) \leftarrow \mathcal{D}(X)$
        yield $X(d_1, \ldots, d_n)$.
    for $x, Y \in dom(\mathcal{D})^2$ do
        $X_{ord} \leftarrow t_{11} < \cdots < t_{1n}$
        $Y_{ord} \leftarrow t_{21} < \cdots < t_{2n}$
        $new\_order \leftarrow \mathcal{O}_2[\![\_ \leftarrow X, Y \mid \{X_{ord}, Y_{ord}\}]\!]$.
        for $(d_1, ..., d_n) \in new\_order$ do
            for $S' \in$ eliminate$((d_1, \ldots, d_n), \mathcal{D})$ do
                yield $S'(d_1, \ldots, d_n) \leftarrow X, Y$.

procedure eliminate($(d_1, \ldots, d_n), \mathcal{D}$)
    for $matches \in$ match_all$(t_1, ..., t_n)$ do
        for $i \in 0 \ldots n/3$ do
            for $S' \in$ remove_all$(matches, i)$ do
                if $S' \in codom(\mathcal{D})$ then
                    yield $S'$

procedure match_all($t_1, ..., t_n$)
    return *all possible ways to match a,b,c triplets*

procedure remove_all($matches, i$)
    return *all possible ways to annihilate i matches*
```

3.1 Grammars

In the grammars below, when applying meta-rules, we consider t_{ij} to be the j-th tuple element of the i-th input state.

G_0: Triple insertion This very simple grammar consists of simply the rules of triple-insertion on a single non-terminal state $\mathsf{W}(x, y)$, as described in chapter 2. Using the $\mathcal{O}$ abstraction, this grammar can be simply expressed as:

$$\begin{aligned}
&\mathsf{S}(xy) \leftarrow \mathsf{W}(x, y).\\
&\mathcal{O}_2[\![\mathsf{W} \leftarrow \epsilon \mid \{a < b < c\}]\!].\\
&\mathcal{O}_2[\![\mathsf{W} \leftarrow \mathsf{W} \mid \{t_{11} < t_{12},\ a < b < c\}]\!].
\end{aligned}$$

G_1: Interleaving words Benefiting from the added expressivity of $\mathcal{O}$, we can extend the previous grammar with a single meta-rule that allows two non-terminals $W(x, y)$, $W(z, w)$ to interact with one another, producing rearranged tuple concatenations and allowing some degree of straddling to be generated.

$$\begin{gathered}
\vdots\\
\mathcal{O}_2[\![\mathsf{W} \leftarrow \mathsf{W}, \mathsf{W} \mid \{t_{11} < t_{12},\ t_{21} < t_{22}\}]\!].
\end{gathered}$$

G_2: Incomplete words This grammar largely expands upon the concept of state interaction of the previous grammar, replacing the idea of triple insertion with the use of non-terminals that correspond to types of incomplete words. There is a very clear and direct correspondence between this grammar and the non-swapping rules of the growth algorithm.

$$\begin{aligned}
&\mathsf{S}(xy) \leftarrow \mathsf{W}(x, y).\\
&\mathcal{O}_2[\![\mathsf{W} \leftarrow \epsilon \mid \{a < b < c\}]\!].\\
&\mathcal{O}_2[\![\mathsf{A}^+ \leftarrow \epsilon \mid \{a\}]\!].\\
&\mathcal{O}_2[\![\mathsf{B}^+ \leftarrow \epsilon \mid \{b\}]\!].\\
&\mathcal{O}_2[\![\mathsf{C}^+ \leftarrow \epsilon \mid \{c\}]\!].\\
&\mathcal{O}_2[\![\mathsf{A}^- \leftarrow \epsilon \mid \{b < c\}]\!].\\
&\mathcal{O}_2[\![\mathsf{B}^- \leftarrow \epsilon \mid \{a < c\}]\!].\\
&\mathcal{O}_2[\![\mathsf{C}^- \leftarrow \epsilon \mid \{a < b\}]\!].\\
&\forall\, \mathsf{K} \in \mathcal{S} \setminus \mathsf{W} :\\
&\qquad \mathcal{O}_2[\![\mathsf{K} \leftarrow \mathsf{K}, \mathsf{W} \mid \{t_{11} < t_{12},\ t_{21} < t_{22}\}]\!].
\end{aligned}$$

Where $\mathcal{S}$ the set of states defined within this grammar:

$$\mathcal{S} = \{\mathsf{W},\ \mathsf{A}^{+/-},\ \mathsf{B}^{+/-},\ \mathsf{C}^{+/-}\}$$

G_3: Universal triple insertion Surprisingly, cases manageable by triple insertion could not be solved by the last grammar despite its elegance. After some thought, we realized that triple insertion is unique in the sense that it can insert three different terminals each at a different position, something that no other pair interaction can directly accomplish. We thus allowed triple insertion back into our last grammar, now at the level of each new state. To achieve this, we added the following universally quantified meta-rule:

$$\begin{array}{l} \qquad\qquad\vdots \\ \forall\, \mathsf{K} \in \mathcal{S} \setminus \mathsf{W} : \\ \qquad \mathcal{O}_2[\![\mathsf{K} \leftarrow \mathsf{K} \mid \{t_{11} < t_{12},\ a < b < c\}]\!]. \end{array}$$

G_4: Refined non-terminals As the peak of our efforts, we replace non-terminals with their refined versions as presented in section 2.3. $A^+(x,y)$ is now split into $A^+_{left}(x,y)$ and $A^+_{right}(x,y)$, with the extra a residing in x and y in each case respectively. To generate this grammar, we use ARIS initialized with the following $\mathcal{D}$ map:

$\mathsf{W} \mapsto (\epsilon,\epsilon)$	$\mathsf{A}^-_r \mapsto (\epsilon, bc)$
$\mathsf{A}^+_l \mapsto (a,\epsilon)$	$\mathsf{A}^-_{l,r} \mapsto (b, c)$
$\mathsf{A}^+_r \mapsto (\epsilon, a)$	$\mathsf{B}^-_l \mapsto (ac,\epsilon)$
$\mathsf{B}^+_l \mapsto (b,\epsilon)$	$\mathsf{B}^-_r \mapsto (\epsilon, ac)$
$\mathsf{B}^+_r \mapsto (\epsilon, b)$	$\mathsf{B}^-_{l,r} \mapsto (a, c)$
$\mathsf{C}^+_l \mapsto (c,\epsilon)$	$\mathsf{C}^-_l \mapsto (ab,\epsilon)$
$\mathsf{C}^+_r \mapsto (\epsilon, c)$	$\mathsf{C}^-_r \mapsto (\epsilon, ab)$
$\mathsf{A}^-_l \mapsto (bc,\epsilon)$	$\mathsf{C}^-_{l,r} \mapsto (a, b)$

3.2 Results

We display three charts, depicting the number of rules, percentage of counter-examples and computation times of each of our grammars for D^3_n with n ranging from 2 to 6. The charts reveal two obvious trends; as grammars get more complex, the number of failing parses steadily declines. This however comes at the cost of rule size growth, which in turn is associated with dramatically exploding computation times. What this practically means is that we are unable to continue testing more elaborate grammars or scale our results to higher orders of n (recall that $\|D^3_n\|$ also has a very rapid rate of expansion).

4 Road to completeness

As the above results suggest, despite our best efforts the problem is yet unsolved. In this chapter we present a collection of unexplored ideas which we

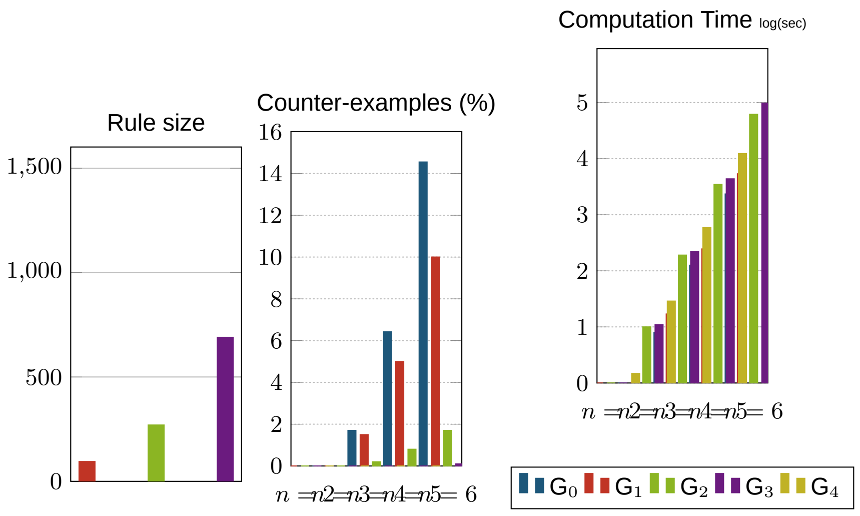

Fig. 3. Performance measures

consider noteworthy. These ideas are related to the concept of formally concluding the completeness of a grammar, bypassing the computational problem of parsing through words in search of an example the tested grammar fails to generate.

4.1 First-match policy and Relinking

Possibly the most intuitive way of checking whether a word w is part of D^3_n is checking whether a pair of links occur that match a_i to b_i and b_i to c_i $\forall i \in n$, where x_i the i-th occurrence of symbol x in w. We call this process of matching the *first-match policy*.

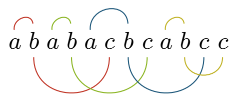

Fig. 4. First-match policy for *"ababacbcabcc"*

The question arises whether a grammar can accomplish inserting a triplet of a, b, c in such a way that the triplet inserted always corresponds to a triplet matched by the first-match policy. If that were the case, it would be relatively easy to generalize this ability by induction to every $n \in \mathbb{N}$. Unfortunately, the answer is seemingly negative; the expressiveness provided by a 2-MCFG does not allow for the rather arbitrary insertions required. On a related note, being able to produce a word state $W(x, y)$ where $w = xy$ and x any possible prefix of w (i.e. all different comma positions i for $i \in 0, 1...3n$, gives no guarantee of being able to produce the same word with an extra triplet inserted due to the straddling property.

However, if rules existed that would allow for match-making and breaking, i.e. match *relinking*, an inserted symbol could be temporarily matched with what might be its first match-policy in a local scope, and then relink it to its correct match when merging two words together. This might be an interesting prospect to look into.

4.2 Growth Rules

The growth rules presented in section **??** are formally sound and complete. Part of them also have a very straightforward correspondence with some of our rules. For the other part, namely the growth rules which produce two edges, there is no direct way of translating them into a 2-MCFG. Adding memory to the grammar by adding states that describe two neighbouring edges might seem appealing but would not solve the problem; two neighbouring edges can in turn interact with their next neighbours in a recursive fashion, which can not be accommodated by our current tuple representation. We cannot, however, rule out the possibility of the growth rules being scalably translatable into m-MCFG rules. In fact, such a translation would be a guarantee of completeness.

4.3 Insights from promotion

The act of promotion translates to cyclic rotation of the A_2 web [**?**]. Perhaps an interesting question here is whether promotion can be handled by a 2-MCFG (as a *context-free rewriting system*). If that is the case, it could be worth looking into the properties of orbits, to test for instance if there are promotions within an orbit that can be easier to solve than others. Solving a single promotion and transducing the solution to all other words belonging to the same orbit could then be a guideline towards completeness. The scale of such a task would of course be much greater, but the combination of these theoretical backbones into a unified body, combined with our mostly applied approach, could shed some light on the problem at hand.

5 Tools

5.1 Grammar utilities

We have implemented the modelling techniques we described in section 2 and distributed a Python package, called **dyck**, which essentially provides the programmer with a *domain-specific language* very close to this paper's mathematical notation. The following example demonstrates the similariry with G_1, defined in section 3:

```
from dyck import *
G_1 = Grammar([
    ('S <- W', {(x, y)}),
    O('W', {(a, b, c)}),
    O('W <- W', {(x, y), (a, b, c)}),
    O('W <- W, W', {(x, y), (z, w)}),
])
```

It ships with several examples, showcasing meta-grammars and ARIS, as well as the complete set of implemented grammars defined in section 3. The package provides a lot of features, relevant to experimentation on m-MCFGs, such as:

- Grammar selection
- Time measurements
- Rule/parse pretty-printing
- Dyck-specific utilities
    - Word generation
    - Soundness/completeness checking
- Grammar (de)serialization

5.2 Visualization

As counter-examples began to grow in size and number, we realised the necessity of a visualization tool to mitigate the challenging task of identifying certain properties they exhibit.

To that end, we distribute another Python package, called **dyckviz**, providing a command-line tool, which allows the simultaneous visualization of tableau-promotion and web-rotation (grouped in their corresponding equivalence classes).

Young tableaux in an orbit are colored via the first-match-policy, which sheds some light on how the *jeu-de-taquin* actually influences the structure of the correponding Dyck words. Interesting patterns have began to emerge, which still remain to be properly investigated.

Webs in a certain orbit, on the other hand, are rendered in a single PDF file (one web per page). Note that we represent them in a slightly simplified way compared to their definition in section 2, ignoring intermediate graph nodes that do not have corresponding non-terminals, as shown below:

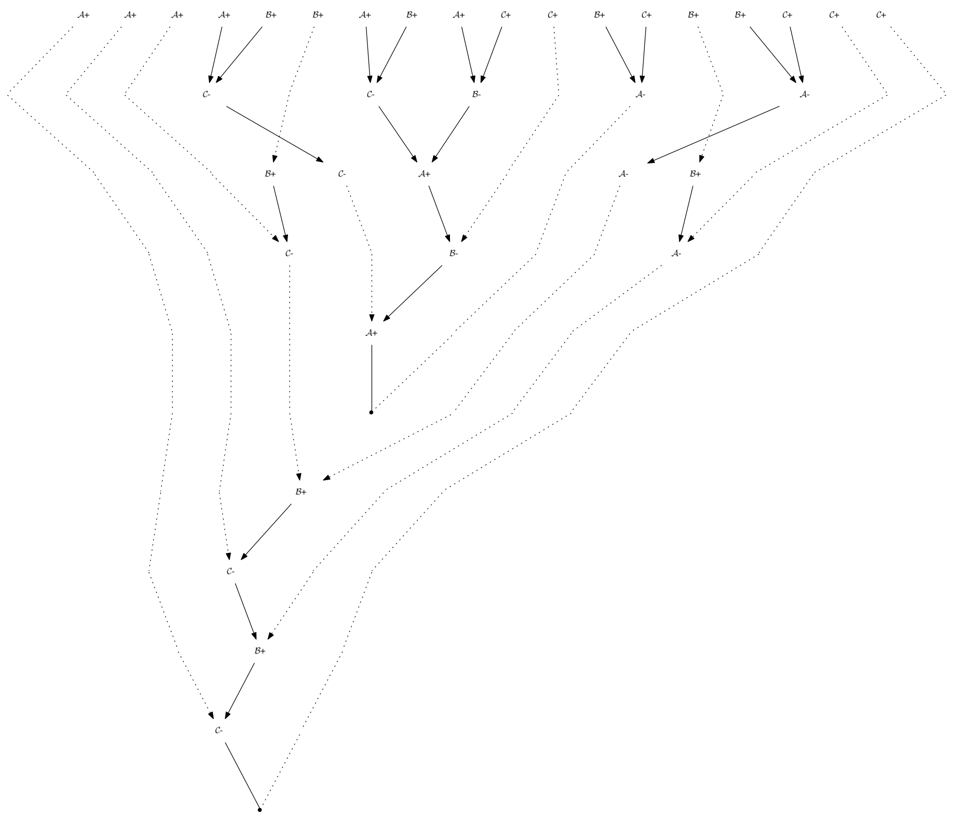

Fig. 5. Spider web of *"abaacbbacbabaccbcc"*

6 Conclusion

We tried to accurately present the intricacies of D^3 and the difficulties that arise when attempting to model it under the scope of a 2-MCFL. We have developed and introduced some interesting techniques and tools, which we believe can be of use even outside the problem's narrow domain. We have largely expanded on the existing tool suite available through the MCFParser and made it easier for it to accommodate MIX-style languages and systems of meta-grammars in general.

Despite our best efforts, the question of whether D^3 can actually be encapsulated within a 2-MCFG still remains unanswered. Regardless, this problem has been very rewarding to pursue, and we hope to have intrigued the interested reader enough to further research the subject, use our code, or strive for a solution on his/her own.

Acknowledgements

We would like to thank Dr. Michael Moortgat for introducing us to the problem and getting us started with valuable bibliography and some initial code.