Procedural Location Generation with Weighted Attribute Grammars

Jan Douwe Beekman
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
l.j.d.beekman@student.utwente.nl

ABSTRACT

In the game development industry, substantial costs are linked to content creation. Procedural content generation is an effective tool for lowering this cost considerably. Many techniques have been developed to generate content, one of which is the use of generative grammars. While grammars have been used to construct the gameplay structures from which a level could be created, it sometimes is perceived as difficult as it required knowledge about grammars that not all game developers have. This method is still promising as it leaves more control to the developer in which complexity or difficulty the resulting dungeon would have. In this paper we discuss the combination of probabilistic and attribute grammars such that the weights used for the probabilities can be computed base on attributes. Whether this combination is useful will be judged by the affected metrics like the complexity of the grammars written in this language.

Keywords

Procedural content generation, Context-free grammars, Probabilistic grammars, Attribute grammars

1. INTRODUCTION

Procedural content generation (PCG) [12] is a technique used to change the content in a game from being designed manually by humans to being generated by an algorithm. When done properly, this could save a lot of time, and therefore costs, and enhance the game significantly in areas such as replayability. PCG is implemented in many ways with different algorithms, one of which is based on probabilistic grammars. Many types of content can be generated but specifically the generation dungeons in role-playing games is heavily explored [15].

Grammars are a way to describe the structure and explain how each part is build from sub-parts. For example, a sentence can have a subject and a subject can contain adjectives. This can also be applied outside of grammars for spoken languages. To give a simple example, a binary tree can be structured like this:

35th Twente Student Conference on IT 2 July 2021, Enschede, The Netherlands.

```
BRANCH ->
    node BRANCH BRANCH |
    leaf
```

This however does not provide the ability to generate a tree directly and a different algorithm would be needed for that. By making a grammar probabilistic, chances will be attached to certain possibilities of structures and thus the grammar could also construct the sentences or other objects. For this example we can add probability distributions like this:

```
BRANCH ->
    node BRANCH BRANCH [weight=1] |
    leaf [weight=2]
```

Now we can generate a branch where it is twice as likely to generate a leaf compared to a splitting node.

1.1 Problem statement

As grammars offer the developers a lot of control over the structure and as the probabilistic side of these grammars also potentially offer control over the outcome according to set parameters, this could be a promising generation method. This method, however, does also have disadvantages such as the fact that not all game developers have the required extensive knowledge of these grammars to efficiently use the technique.

Furthermore, when using probabilistic grammars, it could be a tedious job to determine the probabilities right such that they are not too low and not too high. To get back to the example of the binary tree

```
BRANCH ->
    node BRANCH BRANCH [weight=1] |
    leaf [weight=2]
```

gives on average 2/3 new branches for each branch. This means it will terminate, but has a probability of 67% of resulting in a tree with only one leaf. This can be solved by increasing the chance to create a new node, but when each branch creates on average more then one branch, a branch would expand into infinity. On such a small grammar, these outcomes are easier to see but it is still hard to determine which probabilities are desirable. On bigger and more complex grammars, this problem can get more difficult and solving it could result in an even more complex grammar. Lastly, improvements can be made on this technique to make it easier for developers by giving them more control over the parameters in the dungeon. This could mean, for example, that the developer could easily adjust the difficulty level of a room according to the previous room without creating a huge grammar. Another example would be when a developer wants to increase the size of the level. Adding parameters can prevent going over all the probabilities and changing them accordingly.

1.2 Proposed solution

The most important requirement of the new language is to support structures like:

```
BRANCH(x) ->
    node BRANCH(x-1) BRANCH(x-1) [weight=x] |
    leaf [weight=1]
```

This means that each symbol should be able to have arguments and that these can affect the probability distribution among different options for each symbol. To make the language more intuitive, both arguments and attributes should be implemented. This could make it easier to read and write the data and thus prevent a very long argument list or a very complicated expression to calculate the weight. Arguments will be handled as attributes which means that when these variables are changed, the new value can be read outside of the symbol. Furthermore, the language should still support the basic functionality of any other probabilistic or attribute graph grammar. This means that it can be used to construct objects probabilistically based on weights and that attributes can be calculated. The proposed language will also be a graph grammar and thus each terminal will just be a node. The choice for a graph grammar leads to the user not having to write an extensive tree walker for each grammar they construct. In this paper we will investigate the usefulness of this proposed language with its new functionality.

2. RELATED WORK

Procedural content generation has been applied in multiple ways to dungeons such as space partitioning, agent-based growing, cellular automata, genetic algorithms, generative grammars and many more [12,15]. For these methods, problems and limitations have been identified such as limited control or overlapping structures. Furthermore, like van Rozen and Heijn claim, many problems result in levels not reaching intended goals due to complexity of the grammars. More specifically the impact of small changes to the rules and sheer number of possibilities resulting from recursive rules can cause a lot of bugs [16].

Plenty of research has been conducted regarding grammars in general as well as probabilistic context-free grammars. These grammars can be used to predict complex structures like DNA [17], but are also used to predict which parsing of, for example, a sentences is most likely to be correct when the grammar would otherwise be ambiguous [7]. An example of this would be determining what the sentence "I saw the man with the telescope" means. Is the seeing being done using a telescope or does the man hold a telescope? Probabilistic grammars could help with deciding and thus help with text interpretation.

Using grammars for the generation process of dungeons offers potential as they can prevent issues such as overlapping, unconnected rooms or other structural problem. Some have tried generating the gameplay in this way, while generating the gamespace using complementary algorithms [3, 5, 9, 14]. Others have used grammar rules to insert detail in, for example, a room in a dungeon [10, 16, 19]. Even only combining grammars is shown to be possible [4, 13].

Metrics are useful to objectively reason about programs or grammars. Two important groups of metrics are those being based on size or being based on the flow through the program, like which methods call which or which symbols lead to which other symbols. There are also some other metrics, like for example Halstead effort [8], which has a more complicated calculation to decide how difficult something is to make. There are researchers who investigated useful metrics for grammars. Some of those focused on correctness [1, 18] which is less applicable when there is no clear correctness model. While researching this, focusing on metrics also used for code in programming languages by, for example, viewing symbols as functions is also an option. Lines per method could be number of alternatives per symbol and the number of characters per production rule could be characters per line. Most importantly, there are also metrics which are grammar specific which can be used [2, 11, 20]. Which specific metrics are applicable and used in this situation, will be discussed later in this paper.

3. NEW ARCHITECTURE

3.1 Formal preliminaries

We are about to define *generative weighted attribute grammars.* To do that, let us first recall the classic definition of a context-free grammar (CFG): it is a four-tuple $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{P}, S \rangle$, where $\mathcal{N}$ is a set of nonterminal symbols, $\mathcal{T}$ is a disjoint set of terminal symbols, $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$ is a set of production rules of the form $N \to \alpha$, and finally $S \in \mathcal{N}$ is the starting symbol.

Each CFG implicitly defines a direct derivation relation $\Rightarrow_G \subseteq (\mathcal{N} \cup \mathcal{T})^* \times (\mathcal{N} \cup \mathcal{T})^*$ defined in such a way that $w \Rightarrow_G u$ if and only if $w = w_1 N w_2$ (with $N \in \mathcal{N}$ and $w_i \in (\mathcal{N} \cup \mathcal{T})^*$) and $u = w_1 v w_2$ (with $v \in (\mathcal{N} \cup \mathcal{T})^*$) and there exists $N \to v \in \mathcal{P}$. This relation links each element of $(\mathcal{N} \cup \mathcal{T})^*$(they are called *sentential forms*) with another element that can be derived from it by picking a production rule and replacing its left hand side in the sentential form by its right hand side. Then, the language $L \subseteq \mathcal{T}^*$ generated by the grammar $L = \mathcal{L}(G) \subseteq \mathcal{T}^*$ consists of words (sequences of terminals) that can be derived from the starting symbol in some finite number of steps: $L = \{\alpha \mid S \Rightarrow^*_G \alpha\}$.

Attributes are added to an underlying CFG by adding two more components: the set of attributes $\mathcal{A}$, the mapping that assigns attributes to each node $@ : \mathcal{N} \to \mathcal{A}^*$, and the semantic part σ that associates each production rule $N \to \alpha \in \mathcal{P}$ with a set of attribute evaluation rules, each having the form of $a_0 = \varphi(a_1, \dots a_n)$, where $a_i \in \mathcal{A}$ and φ is some computation function. We are mostly interested in inherited attributes (propagated top-down, the same direction our generation will go), so we can say that $a_0 \in \bigcup\limits_{X \in \alpha} @(X)$ is an attribute of one of the nonterminals of the right hand side of the production rule, and all other $a_i \in @(N)$ are attributes of the node itself. It is trivial to reverse the condition to get to derived (synthesized) attributes, or move to a completely constraint-based setup by imposing no conditions on a_i.

Finally, probabilities are added to our mix by using a weight mapping $\omega : \mathcal{P} \to \mathbb{N}$, computed in such a way that it can take local attribute values into account: $\omega(N \to \alpha) = \psi(a_1, \dots, a_n)$, where all $a_i \in @(N)$.

NB: We note the limitations of our formalisation — such as only using inherited attributes (classic attribute grammars also have synthesized attributes, but those are of much less importance in generative grammars), or not explicitly distinguishing loop-free derivations (again, this only becomes a noticeable issue for ambiguous analytic grammars). However, we ultimately deem them to fall outside the scope of this project. We believe to have left enough freedom in our formalisation to allow extensions: for instance, $\mathcal{A}$ is left undefined, so that one can define it to take the domain of attributes into account.

To summarise:

Definition 3.1. *WAG*
A generative weighted attribute grammar (WAG) is an octuple $\langle \mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{A}, @, \sigma, \omega, S \rangle$*, where* $\mathcal{N}$ *is a set of nonterminals (nodes, classes, types, sorts, ...),* $\mathcal{T}$ *is a disjoint set of terminals (alphabet),* $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$ *is a set of production rules,* $\mathcal{A}$ *is a set of attributes,* $@$ *is a mapping associating attributes to nonterminals,* σ *is a mapping linking semantic evaluation rules to production rules,* ω *is a mapping assigning weights to nonterminals, and* S *is the starting symbol.* ■

3.2 Engineering the new architecture

The new architecture inherits properties of both a grammar language and a programming language. Just like any other grammar language, symbols and finals can be defined. Due to it being a graph grammar language, all the finals will be nodes in a graph. The step towards the programming language comes in the form of scopes. Each symbol and final has its own scope with variables that can be declared and used in computations such as but not limited to the probabilities for deciding the alternative chosen in a symbol.

For the use-case of calculating probabilities, which needs to happen before any assignment or other calculation when the scope of the symbol is entered, the variables or attributes of a scope need to be declared before "calling" or "entering" the symbol. The proposed language supports this by letting the symbols be called with arguments. The values of the arguments can then be mapped to the variables and thus they can be used. These passed arguments can be seen as inherited attributes, but all attributes can be accessed in the parent symbol such that attributes can also be synthesized.

The aforementioned scopes can be seen as a step towards a programming language, but they only make the grammar an attribute grammar. The real programming comes when not only variables change based on other variables, but when also the path through the program can change based on these variables. This is essentially then an if-else statement. In the new language this is possible as the probability of a path can either be increased or decreased and even be set to 0 and thus forcing a path.

3.3 Syntax description

We present a small example to illustrate how the language is implemented. Further examples are given in the project code. [1] For this example, a grammar for a binary tree is written in such a way that most features of the language are shown. Every symbols starts with the symbol name with arguments, followed by all the alternatives which are split with a | character. This means that all the production rules are grouped by the nonterminal. The alternatives have a more complicated structure. First the probability weight is described in square brackets and this is calculated for each alternative when the symbol is entered so all the necessary variables should already be declared in the arguments. After the square brackets, there are three options to follow. These options can be in any order and can be repeated. The order is important as this is the order the program executes the program. The first option is to declare a node. This node should start with a lowercase letter. Secondly, it is also possible to call a symbol. The symbols should start with an uppercase letter and are often followed by arguments in parentheses, but this is not needed. Lastly, there is an option for assignments. These should be declared inside of curly braces. As the example shows, also edges in the graph are declared here using either a ← or → arrow.

```
ROOT(max): [1]
           node
           {node.max = max, node.depth = 0}
           BRANCH(node) BRANCH(node);
BRANCH(parent)
       : [parent.max - parent.depth]
         node_split
         {
           node_split.max = parent.max,
           node_split.depth = parent.depth + 1,
           parent -> node_split
         }
         BRANCH(node_split) BRANCH(node_split)
       | [1]
         node_leaf
         {parent -> node_leaf};
```

4. METHODOLOGY

When a new method for PCG is created, it should be decided whether it is useful or even an improvement compared to other methods. There does not seem to be a benchmark or standardized test for this however [6]. Due to this, it is hard to conclude if the new architecture has any use, which clearly hinders this research. This is the reason why a new benchmark was created.

The new test consists of multiple test cases or goals for grammars which can be implemented in several ways or with several PCG methods. When the implementations were finished, certain metrics can be calculated and reasoned about. These test cases and metrics could be expanded to improve the benchmark. Since this would go beyond the intended scope of the research, the number of cases will be limited to three. Even while this is the case, the results can still be a good indicator of the new language.

4.1 Test cases

4.1.1 Chain

The first test case was a chain of nodes (rooms) with an average of around 20 nodes. The goal was to lower the standard deviation of the amount of rooms. The first solution was to hard-code a number of nodes, the second was using more symbols in the grammar and the third was to use the new calculations for probability. In Table 1, the average number of rooms is described with #Nodes and the standard deviation with σ#Nodes.

4.1.2 Linearity

The second was a grammar which constructs a dungeon with certain decisions a player needed to make to decide a path through the dungeon. The player could encounter a split with 2 branches, a split which later reconnects or a side path which loops directly back to the node where the decision was made. The last option was for the player to have no choice but only one way forward. The goal was a dungeon with around 20 nodes where there were roughly an equal amount of nodes with choices as nodes with a forced path. Again there were 3 solutions. One

[1]Our implementation of this language `https://gitlab.utwente.nl/s1994050/probplusgraphgrammar`

hard-coded, one with more symbols and one with weight calculations. In Table 2, the average number of rooms is described with #Nodes and the number of nodes with a forced path divided by the number of choices is shown under Linearity.

4.1.3 Enemy types

The third test was a "dungeon" with 10 rooms where each room could have one of three types of enemies. The goal was to have only one of the enemies to occur 3 or more times while the enemy in second place occurred twice. This second condition was introduced as it otherwise would be very easy to let the grammar create dungeons with only one type of enemy. Here there was no hard-coded solution where something was fixed. There was one solution with more symbols and one with a simple calculation for probabilities. Furthermore, there was a third solution with more a more complicated calculation that was capable of enforcing the goal 100% of the time instead of 60%-70% which the other 2 solutions managed. In Table 3, the success rate is defined as the percentage of generated graphs where there exists only one type which has 3 or more nodes. Furthermore, when ordered by the number of nodes each type has, the table also displays how much nodes are allocated to the type in the first, second and third place.

4.2 Metrics

In the case of this language, the goal is to see if it makes creating and maintaining grammars easier. This goal should be translated into metrics. Size metrics are useful to measure ease of construct and maintainability as if its smaller, there is less to understand or to type. Complexity metrics are also useful, more computations or possible paths is just simply more to comprehend. Especially Halstead (HAL) effort seems interesting as it consists of both volume and difficulty. The volume is important just like other size metrics are, but the difficulty is even more relevant as game designers needing to much knowledge about grammars was part of the problem statement. Knowledge needed for making a grammar in this language is hard to measure in metrics. This is partially just based on the amount of features the language has, but will also be represented in complexity metrics. To see how complex structures have influences on the outcome, can require just more effort, but it can also require the knowledge on how to use certain complex structures. This, however, is just an assumption. Splitting the effort into these two factors could therefore give more insight in how useful this new language is.

For this research, some metrics seemed to be more useful than others. This is because there were metrics based on the control-flow graph measured. The problem was that the graphs were often only consisting of a few nodes making these metrics close to useless.

5. RESULTS

One of the requirements was that the new language should be able to do what normal other languages support. We found, however, that this language is at least as powerful as any other languages to write probabilistic grammars. Not only is this the case, but the new language is even more powerful. Due to the manipulation of probabilities, a program can be created that chooses an alternative based on data like an if-else statement. This decision is based on read data and the language also support reading data. This, essentially, is all that is needed for a Turing machine. Therefore the language is Turing complete meaning that if the computer which runs the code is infinite, any algorithm that can be written, can be written in this language.

Table 1. Chain length

	Normal	Hard-coded	Symbols	New
#Nodes	20.164	20.906	18.732	20.296
σ#Nodes	19.388	9.276	8.936	10.869
VAR	2	3	4	2
TERM	3	13	3	3
MCC	3	4	8	3
AVS	1.667	4	1.875	1.667
VOL	66.439	289.863	193.588	96.864
DIFF	2.125	2.6	4.5	4.313
HAL	141.182	753.644	871.146	417.726
TIMP	div by 0	100	40	div by 0
CLEV	100	100	100	100
NSLEV	0	0	0	0
DEP	1	1	1	1
HEI	2	2	4	2

Table 2. Linearity

	Normal	Hard-coded	Symbols	New
#Nodes	14.054	20.043	21.366	20.662
Linearity	1.005	1.058	0.911	0.949
VAR	2	3	4	2
TERM	6	6	6	6
MCC	6	6	16	6
AVS	2.833	3.5	2.611	2.833
VOL	474.974	587.555	1403.957	731.1267
DIFF	8.421	8.636	17.6	17.043
HAL	3999.781	5074.338	24709.634	12460.942
TIMP	div by 0	div by 0	40	div by 0
CLEV	100	100	100	100
NSLEV	0	0	0	0
DEP	1	1	1	1
HEI	2	2	4	2

Table 3. Enemy types

	Normal	Symbols	Simple	Complex
Success	11.0%	71.6%	58.2%	100%
#First	4.962	7.252	7.624	6.37
#Second	3.244	1.98	2.096	1.974
#Third	1.794	0.768	0.28	1.656
VAR	2	7	2	2
TERM	13	13	13	13
MCC	4	15	4	4
AVS	3.75	3.2	3.75	3.75
VOL	474.744	1694.335	1134.989	1448.813
DIFF	6.818	17.185	16.667	34.645
HAL	3236.893	29117.463	18916.478	50194.349
TIMP	div by 0	13.333	div by 0	div by 0
CLEV	100	100	100	100
NSLEV	0	0	0	0
DEP	1	1	1	1
HEI	2	3	2	2

This power, however, will technically not make such a big difference in what is theoretically possible. If we take the previous example of a binary tree, then we can simply see that, maybe with a lot of hard-coding, the result is also possible without probability calculations. The way this can be done, is by creating new symbols for each possible value of the arguments like:

```
BRANCH_TWO ->
    node BRANCH_ONE BRANCH_ONE [weight=2] |
    leaf [weight=1]
BRANCH_ONE ->
    node BRANCH_ZERO BRANCH_ZERO [weight=1] |
    leaf [weight=1]
BRANCH_ZERO ->
    leaf [weight=1]
```

With the example of a tree, it is clearly sometimes easier and most definitely shorter to use arguments to calculate the weights, although other options are still possible. In this example, the constructed grammar can even be as short as the descriptive grammar which cannot generate a tree. This is why also in practice the new language is far more powerful as nobody would hard-code all the options of what the arguments could be in a more complex grammar.

5.1 Experimental Results

Like previously described, for each test case, multiple implementations were constructed. First, a grammar was made that only described the structure such that it could theoretically create all the outcomes all the other grammars could make. This grammar partially acted like a control-group by giving more meaning to all the other measurements.

After that, implementations were needed that did not require the extension with computable probabilities. The easiest way to make sure outcomes like the number of nodes result reach the goal, is by hard-coding. Going back to the running example, this would result in the first split being hard-coded which effectively doubles the tree size.

```
ROOT ->
    node BRANCH BRANCH [weight=1]
BRANCH ->
    node BRANCH BRANCH [weight=1] |
    leaf [weight=2]
```

A different way of making sure the outcome reaches the goal is by implementing more steps or routes towards the result. In the binary tree example, a solution was mentioned to decrease the probability of splitting each step. This would require a lot of symbols, but a compromise could be made with having a symbol with a probability of splitting into itself, while also having a chance of splitting into the next symbol. This next symbol could split into two branches while also having a chance to terminate into leaves. This would also increase the average size while decreasing the chance of the tree only containing one node.

```
BRANCH_START ->
    node BRANCH_START BRANCH_START [weight=1] |
    BRANCH_END [weight=2]
BRANCH_END ->
    node BRANCH_END BRANCH_END [weight=1] |
    leaf [weight=2]
```

Furthermore, the goal could be reached by adding more paths in the grammar. This would mean that at the start, a decision is made to go for a certain outcome. These options could be, for example, a symbol that constructs a very big tree while there is also an options which creates a small tree. Although this does not achieve a lot in the case of binary trees, it might result in outcomes the developer wants.

```
ROOT ->
    BRANCH_BIG [weight=1] |
    BRANCH_SMALL [weight=1]
BRANCH_BIG ->
    node BRANCH_BIG BRANCH_BIG [weight=1] |
    BRANCH_END [weight=1]
BRANCH_SMALL ->
    node BRANCH_SMALL BRANCH_SMALL [weight=1] |
    leaf [weight=5]
```

Lastly, there was also the implementation which used the different weight calculations. These grammars could show whether measurements would be changed. Like mentioned in the test case descriptions, the third test also had this option split into a simple and a complex grammar to show the power of the extension for probabilistic grammars.

5.1.1 *Measurements*

Next to the previously discussed measured numbers, each table of a test case also has standardized metrics. The important metrics are the number of nonterminals (VAR), the number of terminals (TERM), the McCabe complexity (MCC), the average right-hand side (AVS), the Halstead volume (VOL), the Halstead difficulty (DIFF) and the Halstead effort (EFF). The McCabe complexity, or cyclomatic complexity, was measured as the total number of alternatives in a grammar, just like Power et al. [11] count this complexity. Furthermore, also the tree impurity (TIMP), the normalized count of levels (CLEV), the number of non-singleton levels (NSLEV), the size of the largest level (DEP) and the maximum height (HEI) are measured. These, however, are not as relevant as they contain information on the control flow graph, which in these cases often only consists of two nodes.

5.1.2 *Size Metrics*

Just like the expectations, the number of terminals, nonterminals and alternatives are lower or equal on all the grammars constructed with the new feature of argument based weights. This is probably because the “normal” grammar which can only properly describe the outcome instead of creating the wanted outcome, can be adjusted in probabilities without adding anything else. In the cases where the results were equal, this might be because the grammars are small and therefore the outcomes are small. Therefore further expansion of test cases would improve this benchmark. This leads us to conjecture that size metrics improve with this new feature of probabilistic grammars, but a more detailed study needs to be conducted with a larger sample size and with bigger constructed grammars.

5.1.3 *Halstead complexity*

For a grammar where certain structures are fixed, the expected volume would be higher while the difficulty is still low. The reason for this is that hard-coding decreases the possibilities and calculations by replacing those with assignments. Meanwhile, a grammar with more symbols results in both a high volume and a high difficulty as there are more or longer paths and each part of the path has a certain volume and difficulty which are accumulated in the total. Lastly, for a grammar with more calculations, we expect the difficulty to go up drastically due to the need for more math, while the volume only increases partially to declare all the terms needed in the calculations.

Even though the results are lacking in quantity, it is interesting to see that the results show the same as the expectations. Especially comparing the options with more symbols or more computations, it is interesting to see that, in all these cases, the difficulties are roughly equal. There is, however, still a difference in volumes and thus we can conclude the new feature for calculation weights, most likely, requires less effort when creating grammars. The results also show that the option with hard-coding structures can still have the lowest effort. In the second test case this is shown, but as the chain length case shows, this is not always the case. However, some developers might not even want to force this in the results, so the recommendation for minimizing the effort would depend on the programmers preferences.

Furthermore, when the probability calculations are used, it is possible to achieve more. For example, in the test-case about enemy types, the complex solutions could achieve a success rate of 100%. The effort to create such a grammar will be significantly higher. So again, it is up to the developers preference to decide which type of solution to use.

5.1.4 Control-flow graph

Like mentioned earlier, metrics based on the control-flow graph are less useful as the graphs were so small. Only the height has an explainable difference because the grammars which had more symbols, also had a bigger height.

In other metrics there were clear signs how little the results meant. In the tree impurity calculations, for example, there were a lot of divisions by zero, which shows that the calculation is not meant for such small numbers.

6. CONCLUSIONS

This research suggests that there are actual use-cases for this addition of probabilistic grammars. This is mostly due to the fact that it makes more possible to practically construct while also decreasing the size of other grammars. The best indicator for this is that one of the test cases, the new language made it possible to get a 100% success rate since it was a lot easier to set limits.

The new method for PCG does also have negatives as the difficulty of constructing grammars does not decrease. This is both because the constructed grammars itself do not seem to be less difficult, but also due to the simple fact that more features of a language results in more to learn and more options to consider while programming. This, however, does not mean it would be useless. The addition of the weight calculations can be seen as a tool in the toolbox of a game designer, for which the developer itself can choose whether to use it or not.

Given that the objective was to get a method of creating content which did not require a lot of knowledge from the developer, the Halstead difficulty was more important than the Halstead effort. Given that these difficulties tend to be equal or worse, this means that the results do not conclude that new language is ”better”.

In this research, it is important to notice that the amount of evidence is small and it would be useful to extend it by testing on a larger group of developers with more test cases. This would lead to more convincing data. This does not take away that this research shows that the extension on probabilistic grammars is interesting as there are clear cases when the solution is easier to construct.

7. REFERENCES

[1] S. Bangalore, O. Rambow, and S. Whittaker. Evaluation Metrics for Generation. In *Proceedings of the First International Conference on Natural Language Generation, Volume 14*, INLG ’00, page 1–8. Association for Computational Linguistics, 2000.

[2] E. Csuhaj-Varjú and A. Kelemenová. Descriptional complexity of context-free grammar forms. *Theoretical Computer Science*, 112(2):277–289, 1993.

[3] B. De Kegel and M. Haahr. Towards procedural generation of narrative puzzles for adventure games. In R. E. Cardona-Rivera, A. Sullivan, and R. M. Young, editors, *Interactive Storytelling*, pages 241–249, Cham, 2019. Springer International Publishing.

[4] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(03):216–228, jul 2011.

[5] A. Gellel and P. Sweetser. A hybrid approach to procedural generation of roguelike video game levels. In *International Conference on the Foundations of Digital Games*, FDG ’20, New York, NY, USA, 2020. Association for Computing Machinery.

[6] A. Gellel and P. Sweetser. A hybrid approach to procedural generation of roguelike video game levels. In *International Conference on the Foundations of Digital Games*, FDG ’20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] S. Geman and M. Johnson. Probabilistic grammars and their applications. In J. D. Wright, editor, *International Encyclopedia of the Social I& Behavioral Sciences (Second Edition)*, pages 9–15. Elsevier, Oxford, second edition edition, 2015.

[8] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[9] J. Kowalski, R. Miernik, P. Pytlik, M. Pawlikowski, K. Piecuch, and J. Sękowski. Strategic features and terrain generation for balanced heroes of might and magic iii maps. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[10] K. Merrick, A. Isaacs, M. Barlow, and N. Gu. A shape grammar approach to computational creativity and procedural content generation in massively multiplayer online role playing games. *Entertainment Computing*, 4(2):115–130, 2013. cited By 7.

[11] J. F. Power and B. Malloy. A metrics suite for grammar-based software. *J. Softw. Maintenance Res. Pract.*, 16:405–426, 2004.

[12] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games.* Computational Synthesis and Creative Systems. Springer, 2016.

[13] G. Smith, M. Treanor, J. Whitehead, and M. Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the Fourth International Conference on the Foundations of Digital Games (FDG)*, pages 175–182, 2009.

[14] R. van der Linden, R. Lopes, and R. Bidarra. Designing procedurally generated levels. In *AAAI Workshop - Technical Report*, pages 41–47, 10 2013.

[15] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.

[16] R. van Rozen and Q. Heijn. Measuring quality of grammars for procedural level generation. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, New York, NY, USA, 2018. Association for Computing Machinery.

[17] G. W. Wilburn and S. R. Eddy. Remote homology search with hidden potts models. *PLOS Computational Biology*, 16(11):1–22, 11 2020.

[18] C. Yang. Rage against the machine: Evaluation metrics in the 21st century. *Language Acquisition*, 24(2):100–125, 2017.

[19] R. Zmugg, W. Thaller, U. Krispel, J. Edelsbrunner, S. Havemann, and D. Fellner. Procedural architecture using deformation-aware split grammars. *Visual Computer*, 30(9):1009–1019, 2014. cited By 5.

[20] M. Črepinšek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, and G. Roussel. On automata and language based grammar metrics. *Comput. Sci. Inf. Syst.*, 7:309–329, 05 2010.