EvoPoem: Context Free Grammars for Automated Poetry Generation

Jan de Mooij 3966615
First assessor: dr. G.A.W. Vreeswijk
Second assessor: dr. R.W.F. Nouwen

28 February 2016
Bachelor Artificial Intelligence, Utrecht University
7,5 ECTS Bachelor Thesis

Chapter 1 Abstract

In this thesis, we discuss various poetry generators. We present our own text synthesis and natural language generation algorithm that can be used with various machine learning technologies, as well as a setup for the final machine learning enabled poetry generator, which we call EvoPoem. The algorithm is able to produce short, grammatically correct sentences, and create a visual spacing that suggests rhythm. This algorithm uses a grammar, a lexicon and a feature and unification algorithm enriched with constraint satisfaction, to parse a string of bits deterministically into a potential poem.

Table of Contents

Chapter 2 Introduction

In 1961 the book *Cent Mille Milliards de Poèmes*, a book by the French writer and poet Raymond Queneau, was published. The book contains ten sonnets, of which each of the fourteen lines are printed on a separate strip of paper, so that fourteen strips make up one page. The reader is free to combine the lines from any of the poems to reveal a new, original poem. This way, the book effectively contains 10^{14} poems and thus shows how, even with a small number of building stones, a very large number of poems can be created. This idea opens up a lot of possibilities for generating poetry with artificial intelligence. A simple search on websites such as the search engine Google or the social media website and creative platform Tumblr, show a wide variety of poems written by hobbyists. Many of these poems consist of a few short lines, where indentation is used to suggest rhythm. Many of these poems do not incorporate rhythm or rhyme and, in that respect, differ very much from Shakespearian sonnets, but attract a lot of attention by fellow internet users nonetheless. We are interested in an algorithm that could write this kind of poetry, and learn from reader feedback to write increasingly good poems. This kind of poetry has the added advantage that some grammatical errors may be allowed, as the poem is generally judged by its emotional impact on the reader.

We are interested in the core mechanisms of a system that can write poetry and learn to write better poetry, without focussing on the machine learning process. We focus on the core system only, which should be able to generate a few short, grammatically correct lines, including indentation. This led to the following question: What is needed to create a text generator that uses a modular grammar and lexicon which can easily be extended, to create texts that can be used for machine learning to create poetry?

We will produce a text synthesis and natural language generation algorithm for poetry-like texts. We will also shortly discuss techniques that could be used with our algorithm to create the final machine learning enabled poetry generator, which we call EvoPoem, although we will not create the generator at this time.

Speech and text generation is a widely discussed topic within the field of artificial intelligence, so we believe this topic fits well within the field. Although our algorithm will not be able to produce text with intentional meaning, we believe it will be able to produce text that *appears* to be meaningful, when adjoined with an appropriate machine learning algorithm.

Although implementing the actual machine learning algorithm – as well as having humans rate the poetry – falls outside of the scope of this thesis, the concept offers a stepping stone into the other branches of the field of artificial intelligence. Due to the nature of machine learning, a goal must be provided that can be expressed as a mathematical formula, before machine learning can take place. Using machine learning on art is often not trivial, as a score indicating the quality of art is subjective and therefore hard to capture with mathematics. A mechanism that allows machine learning to work on poetry generation could therefore be a valuable addition to the field. Additionally, if such a system would be able to produce proper poetry by learning from user feedback, it would provide useful insights to the field of psychology, as this may offer understanding of how willing humans may be in accepting mistakes when dealing with a machine. Furthermore, it raises the philosophical question whether art can only exist when the message it carries is created explicitly and with intent by the artist, or if it may also be called art when meaning is only perceived by the beholder.

This thesis describes the proof of concept we developed for a simple poetry text generation algorithm that meets the requirements we mentioned earlier. First we will look into similar attempts in this field, and how they differ from what we wish to achieve. Then we will describe the goal of this research in more detail and explain our methods. We will then present the proof-of-concept algorithm we developed

as the core mechanism for this poetry generator, and discuss the performance of this algorithm. We will finish with a discussion of our methods and with suggestions as to how this algorithm can be used in further research.

Chapter 3 Theoretical Context

As a research interest, the topic of generating art – and poetry in particular – is not new. A good overview of previous attempts of poetry generation is presented by Oliviera (2009). For example, PO-EVOLVE is a prototype implementation of a model for poetry generation using evolutionary algorithms and neural networks (Levy, 2001). Levy compares creative processes to Darwinian evolution, where something good that is also something new, is created by accident. This model of creativity served as the inspiration for his model of poetry generation. In the PO-EVOLVE implementation, limericks are created by randomly generating a population – a series of candidate solutions to a search problem, in this case texts – with a set of words, which are tagged with phonetic and stress information and which are then placed on a rhythm template. The words at the end of each line are selected to rhyme. Evaluation happens by means of a recurrent neural network, to which the words of the poem are fed one by one, and which then returns a score indicating the creativity of the poem based on pre-defined rules.

Another example is *McGonnagall* (Manurung, 2003), an advanced poetry generator developed by Manurung for his Ph.D. and based on his previously developed chart system (Manurung, 1999). This system makes use of genetic algorithms as well. A grammatically correct poem is constructed and can be evolved by replacing, adding or removing parts of the parsing tree. The goal of this evolution is to create poems which are syntactically correct and fulfil a certain goal (their given example is *"to create an iambic pentameter couplet that describes the act of walking and sleeping by a person named John*") (Manurung , 2003, p. 56). A start sentence ("*John walks*") is generated based on the semantics of the goal, and is expanded by the algorithm to become a poem. His most important addition to the field, however, seems to be his definition of what a poem is. As they point out, a poem can take many forms and a circular definition such as "A poem is a poem because people call it a poem", while complete, does not allow for falsification of whether the generated poem is actually a proper poem (Manurung, 2003, pp. 15-16). Therefore, they created three traits that a text must satisfy before it can be classified as a poem:

1) **Meaningfulness**: A poem must convey a (conceptual) message and this has to be an *intentional* message
2) **Grammaticality**: A poem must abide by the rules of grammar for the given language and be very restricted in its use of *poetic license*.
3) **Poeticness**: A poem must exhibit poetic features, such as rhythm or rhyme.

Although Manurung points out this definition excludes a lot of texts that would be classified as poetry by the previous (circular) definition, these three traits seem to be treated as rules by many researchers who try to create a poetry generator. In his text, Oliviera even uses these traits to assess the different poetry generators he discusses on their overall performance (2009).

Of these three traits, the issue of meaningfulness seems to present the most difficulties. Meaningfulness is usually achieved with a semantic modelling of words (Levy, 2001; Manurung, 2003) or a semantic relation map (Toivanen, Toivonen, Valitutti, & Gross, 2012). In the first case, the semantics are being build and then translated to natural language, while the latter mechanism only uses words related to a given topic.

The second trait, grammaticality, is ensured by a variety of techniques. Some authors use some form of text generators (such as context free grammars or chart parsers) (Manurung, 2003) while others use structures of existing poems, where the existing words are partly, or indeed completely, replaced with words matching a given lexical category (Toivanen et al., 2012). Still other authors use their application

for translating poems from other languages. This method deals with the matter of meaningful content, as meaning is already present in the original poem (Greene, Bodrumlu, & Knight, 2010).

Most of the poetry generators seem to focus primarily on the concept of poeticness, however. It seems that as long as a poem has the proper poetic form, content is treated as less relevant. To ensure poeticness, end of line words are matched to rhyme, and words in a sentence are assessed and selected on how their metre (rhythm) fits into a sentence. Emphasis information for words can easily be retrieved from a dictionary, and metres for sonnets are well defined. Some methods use constraints to ensure a rhythm (Manurung, 1999), others make use of a predefined rhythmic scheme alongside their grammatical template (Greene et al., 2010).

Although not all discussed systems try to incorporate all three traits mentioned in Manurung's definition of poetry, most of the above discussed systems at least require their generated poetry to exhibit poetic features (i.e. rhyme and rhythmicality). This is probably mainly to distinguish from automatically generated narrative: in poetry, there is a strong connection between content and form and, as some may argue, to the extent where form is more important than content (Toivanen et al., 2012).

In some cases, however, following Manurung's definition seems to be a purely pragmatic choice. As Manurung states, the hardest part of automatically generating creative works, is evaluation in such a way that the assertion that the output of a given system that generates poetry can be verified as being poetry. In other words, the assertion must be falsifiable (Manurung, 2003, p. 3). So a strong definition is needed, to assess the performance of a generator. If no such strict rules for evaluation are at hand, the only logical alternative seems to be using something like a Turing test, or acceptance of the poem in some established venue (Toivanen et al, 2012, p. 178). This is exactly the approach that Toivanen et al. took and which seems to distinguish their research from other research on this topic. To assess the performance of their poetry generator, they asked a human jury to rate their automatically generated poems on the following aspects (2012, p. 178):

1) How typical is the poem?
2) How understandable is the poem?
3) How good is the language?
4) Does the text evoke mental images?
5) Does the text invoke emotions?
6) How much does the subject like the text?

This results in a score for the poem that is probably more subjective, but accounts better for the creativity of a poem than Manurung's definition does, as it does not limit the rating of a poem to a constraining definition of what a poem is.

There are many attempts at generating poetry, using different degrees of artificial intelligence and automatic text generation. Manurung's work has had a great influence in the field with his definition of what a poem is. For him and many other researchers, rhythm and rhyme are such important aspects of poetry that the quality of a poem can be judged by them.

Chapter 4 Method

With the final poetry generator EvoPoem, we aim to create a system that learns to generate increasingly good poems by means of a genetic algorithm. Each poem will consist of a few short sentences and will be presented using various degrees of indentation. The quality of the poem is determined by means of human ratings. The grammatical structure, the word choice, and the visual spacing are subject to a genetic algorithm. This means all these elements, which constitute the poem, can evolve throughout the process. If enough poems are generated, and rated by real humans, we hope EvoPoem will output increasingly good poetry which feels more natural in rhythm and meaning than their predecessors, and which feels more appealing in their visual presentation. In this thesis, we focus on the underlying mechanisms of representation and the generation of single candidate poems, rather than on the machine learning process.

The internal representation of a poem in the EvoPoem generator will be a bitstring: a sequence of ones and zeros which are given meaning by a generator. The generator must be able to parse this bitstring to a poem, and must be able to do so deterministically, meaning each bitstring has a one to one mapping with a poem. It is important that every time a specific bitstring is parsed, the output will be the same, because the evolution happens based on the bitstring, and not on the poem itself, which is rated by the user.

To give the bitstring meaning for the parser, the bit string must be converted to a sequence of integers. The length of the bitstring may vary by implementation. The bitstring must be read as a stream, where some amount of bits form a unit together, which is converted to an integer. If the end of the bitstring is reached, the bitstring is read from the start again. As units consisting of multiple bits are used, the integer output is only repeated if the length of the bitstring is divisible by the unit length.

In order to have the bitstring parsed into a grammatically correct poem, a grammar needs to be defined that can generate a template which holds the grammatical structure. This template needs to be filled in with words that fit the template, but which are also congruent with each other. For example, a template could be:

Noun → *Verb* → *Det* → *Noun*

This template could be filled correctly with the following words:

I walk a dog

The template could also, however, be filled with these words:

* I walking an dogs

A method must be found to avoid ungrammatical sentences like the last example. This method must select a word that fits the grammatical template and then select an inflection of that word which is congruent with the rest of the sentence in which it is used.

To present the poem in a visually appealing manner, a way must be found to insert indentation to the poem. This too must happen based on the bitstring which the poem is generated from, and must be done deterministically.

To achieve all this, a proper data structure for the grammar, the lexicon and the poem itself have to be designed. For the purposes of this proof of concept, we will use a very small grammar and lexicon, but this method must work with large grammars and lexicons just as well. In other words, the method must be easily extendible, so the algorithm itself requires no rewriting when working with a different or more extensive grammar or lexicon.

Considering the baseline representation of a poem is a bitstring, and this bitstring can be parsed deterministically to a poem, given a certain grammar and lexicon, the evolutionary process is not implemented in this proof of concept. A genetic algorithm to actually evolve the generated poetry is fairly simple to implement as the workings of genetic algorithms are documented very well (Mitchell, 1998). Furthermore, the method of generating new poems can be changed very easily from a genetic algorithm to something else, e.g. an artificial neural network, without too much work, because the poem is parsed from a bitstring. Any machine learning algorithm used in EvoPoem only requires to create valid bitstrings, based on the user rating. For the purpose of testing the poetry generator, we will use randomly generated bitstrings.

Chapter 5 Novelty of Our Approach

All attempts of automatic poetry generation that we have reviewed in Chapter 3 use a well constrained definition of what a poem is. The generation process is bound by constraints, to ensure meaningfulness, poeticness, and grammaticality. All generated texts must follow a strict metre to qualify as a poem. Some generators use existing poems to replace words until the meaning of the poems subject is the one envisioned by the generator, others only use the grammatical structure of existing poems and fill in their own words. However, all of these generators have in common that the creative freedom of the computer is hugely limited by the set definition.

We propose to let go of the restrictions of poeticness and meaningfulness so this will not limit the search space of possible poetry that can be generated based on user ratings. Rather than letting an algorithm define what constitutes proper poetry, we intend to have proper poetry shaped by the ratings given by readers of the poetry. The restrictions of poeticness and meaningfulness are too constricting for these purposes. We intend to keep the constraint of grammaticality in place, as this hardly limits the search space for a possible poem, while being a relatively easy first filter to apply on generated poetry. To ensure grammaticality, we propose to use a customized context free grammar (CFG), which we will elaborate on in Chapter 6.

We further propose making use of a user rating system to assess the fitness of a generated poem, rather than programmatically checking the satisfaction of predefined and possibly limiting constraints. The exact implementation of such a user rating system is beyond the scope of this research project, but our algorithm was developed with a relative rating system in mind. Such a rating system could consist of a website that allows users to select the better of two poems that appear simultaneously on the screen and replacing the lesser of the two with a new and as yet by that user unrated poem. This way of rating poetry is inspired by the Turing test, which, according to Toivanen (2012, p. 178) has been suggested as a way to evaluate poetry before. Certainly, because of the subjective nature of art, a human rating system will work very well for our approach. If the user base is large enough, this further ensures that the poem is determined to be good in general and by a large audience, rather than being good according to predefined and potentially incomplete mathematical formulas, or according to a few individuals with a very specific taste in poetry. Especially with the type of poetry that will be generated with EvoPoem, which may not appeal to poetry critics, but hopefully will appeal to a large audience of poetry enthusiasts, this way of rating poetry would work especially well. This approach does, therefore, not require a more strict definition of poetry than “A poem is a poem, because people call it a poem,” (Manurung, 2003, p. 7) but in fact embraces this definition.

This approach differs from previous cases, because poeticness and meaningfulness are not ensured by the algorithm, but introduced by the genetic algorithm that uses user rating as a fitness function. Previous attempts focus on generating one genre of poetry, such as Shakespearian sonnets or Haiku’s. In contrast, EvoPoem will very likely not output just one specific type of poetry, nor will poetry be generated with meaningfulness in mind. This is something that Manarung condemns, but which may yield very interesting insights into the previously discussed philosophical and psychological questions.

Chapter 6 Construction of the Algorithm

For McGonnagall (Manurung, 2003), a reversed chart parser was used to generate grammatically correct poetry. The evolution process is performed by replacing parts of the parser tree in the poem. We propose to use a context free grammar (DCG) (Jurafsky & Martin, 2009, pp. 421-426) for the generation of the grammar template. The approach is much the same as that of definite clause grammars (DCGs) that have been created for use with Prolog (Jurafsky & Martin, 2009, p. 560). The grammar defines nodes, which may contain references to other nodes or to leaf nodes. The end nodes, or leafs, are a specific class of words, such as a noun or a verb. The root node will be a sentence, denoted as S. This node consists of other nodes, for example a noun phrase and a verb phrase. The noun phrase may consist of a determiner and a noun, and the verb phrase could be just a verb. According to this CFG, the only grammatically correct sentence that can be generated is one of the form *determiner* → *noun* → *verb*. In many grammars, however, each node, except for an end node, offers a choice for different phrases. A verb phrase may as well consist of a verb *and* a noun phrase. These choices can be represented as a tree. In our approach, we use the next integer returned from the bitstring to select which branch of the tree to expand, implementing grammatical evolution (Dempsey, O'Neill, & Brabazon, 2009, pp. 9-24). This way, as long as the grammar does not change, a bitstring will always result in the same grammatical template. This approach is very suitable to use CFGs to parse bitstrings, which means that anyone who wishes to adapt our system to generate poetry in their own way, only has to generate a bitstring instead of having to redesign the grammar, lexicon and text generation algorithm.

We implemented this grammar in Java as a search tree which wespecifically adapted for our purposes. Each node in this search tree is flagged as either *group*, *clause* or *lexicon*. A *group*-node defines a split point, and contains one or more nodes, one of which can be expanded. The example of the verb-phrase, which can either be a verb, or a verb and a noun phrase, is a *group*-node. A *group*-node indicates the current node has to be replaced by exactly one of its members. A *clause*-node contains clauses, or phrases. The node for *sentence* as well as the given examples for the noun and verb phrases are such *clause*-nodes. It indicates the current node has to be replaced by *all* its members. Typically, a clause node will contain a group node and vice versa. The last group of nodes are the *lexicon*-nodes. They contain a reference to exactly one word class (e.g. noun, verb, determiner) and indicate the current node has to be replaced by one word from the given word class.

Figure 1 shows the pseudo algorithm for generating the grammatical template from the bitstring, using the grammar as described above.

```
1.  Let grammar ← GrammerRootNode
2.  Do:
3.      For each x in grammar:
4.          If type of x == clause:
5.              let x ← x.members
6.          Else if type of x == group:
7.              Let y ← next_bit_string_int()
8.              Let x ← x.member(y % x.size)
9.          Else:
10.             Continue
11. While not all x in gram of type lexicon:
12. return grammar
```

Figure 1 The pseudo code describing the process of generating the grammatical template from the bitstring

The lexicon is represented as a list of word classes. Each word class contains a list of objects of words that fit into that class. The word object contains the base form for a word, and a list of possible inflections. Each end node in the grammar points to exactly one word class which provides a list of words that fit into that position of the grammatical template. The algorithm from Figure 1 returns a grammatical template that contains only lexicon-nodes. From each of those lexicon-nodes, one word is selected. Figure 2 shows the pseudo code for selecting the words to fill the grammatical template. It takes the grammar from the algorithm shown in Figure 1 and the bitstring as arguments and returns an ordered list of word objects.

```
1. Define empty wordlist
2.         For each x in grammar:
3.                 Let lexicon ← x.getLexicon
4.                 Let y ← next_bit_string_int()
5.                 Wordlist.add(lexicon.get(y % lexicon.size))
6. Return wordlist
```

Figure 2 The pseudo code describing the process of selecting words to fill the grammatical template

Again, because the words are selected based on the next integer returned by the bitstring, the choice of words is deterministic, given that the grammar and lexicon remain unchanged. The final challenge remains to pick the correct inflection for each word in order to ensure congruence between words. One option would be to tag the initial CFG with extra information about, among other things, person, number and tense, but this would mean creating a separate clause for each possible inflection and would result in unmanageably large grammars. Instead, we opted for another approach, suggested by Jurafsky and Martin (2009, pp. 523-561) which uses features and unification. We provide each word inflection with an attribute value matrix (AVM). This AVM contains attributes that are relevant for the word class, such as person, number or tense. Not all attributes are relevant for each word class; e.g. tense is not relevant for a noun, so only attributes relevant for the word are used. The AVM as proposed by Jurafsky and Martin is then adapted to allow features to have multiple values. This way, a word object – which sits one abstraction level above the inflection object – can contain the combined values of an attribute of all inflections of that word. The grammar is then extended so that additional constraints can be defined on unification within a clause. This ensures not all features are necessarily unified, which is especially useful in subordinate clauses, where a noun may appear that is not congruent with the subject.

Now that the structure is in place, the algorithm for unification needs a few adaptations as well to make use of the previously made changes. First, unification must unify all possible values for a feature, rather than merely unifying all features, to ensure the set of possible inflections remains as large as possible. This adaptation is necessary to generate text, instead of parsing text as the algorithm as described by Jurafsky and Martin was originally intended for. Inflections have to be selected based on how they fit within the structure. One possibility would be to choose one inflection that fits for a single word and work from there, starting over if a subsequent word has no inflections that fit into the sentence. This could be done with depth-first or breadth-first search. Instead, we opted to convert the search problem into a constraint satisfaction problem, where constraint propagation is used to search through a domain of possibilities (Russel & Norvig, 2013, pp. 202-230). This means our method starts with the complete domain of possible inflections for each word. With every step, all inflections that are no longer congruent with a chosen inflection are eliminated from subsequent words. The result of this approach is that during unification, a feature appearing in both AVMs will return the conjunction of the domain of the feature in both AVMs. The rest of the rules for unification remain unchanged.

The nodes in the grammar do not require the features, as the features are only used to select the proper inflection after the final word list is created. This means unification can happen after the final list of words is generated by the algorithm in Figure 2. However, clauses are used to define which attributes of two AVMs must be unified. To do this, whenever a clause is encountered, the unification instructions have to be unified with the children of a clause node, for example a clause for *noun phrase* may consists of a determiner and a noun, and declares unification must happen on the feature *number*. The clause *noun phrase* is then replaced with the lexicon nodes *determiner* and *noun*, both of which receive a reference to each other's feature *number* in their AVM. After the final word list is generated, each word has an AVM that describes all allowed values for its attributes. Where two features have been unified, the two AVMs point to the same object, to ensure they remain similar, even after changes have been made to either. This is just as proposed by Jurafsky and Martin (2009, pp. 525-526). The inflections are now chosen by iterating over each word, populating the domain with a list of all inflections for the word that could be unified with the AVM for that word, and then using the next integer from the bitstring to select an inflection from the domain. Then, the AVM of the inspected word and the selected inflection are unified and the values from the domain that are eliminated in this step are eliminated from all subsequent words using constraint propagation. This ensures inflections that can be selected for future words remain congruent with all earlier selected inflections.

Figure 3 shows the algorithm, which takes the output from the algorithm of Figure 2 as an argument, and selects one of the allowed inflections of a word, ensuring congruency between words. This algorithm then returns a grammatically correct single sentence.

```
1.  Define empty line
2.  For each x in wordlist:
3.          Define allowed_inflection_list
4.          For each inflection in x:
5.                  If inflection.AVM.unify(x.AVM):
6.                          allowed_inflection_list.add(inflection)
7.          Let y ← next_bit_string_int()
8.          Let selected ← allowed_inflection_list.get(y % allowed_inflection_list.size)
7.          Line.add(selected)
8.          X.AVM.unify(selected.AVM)
9.  Return line
```

Figure 3 The pseudo code that shows the procedure for selecting congruent inflections for each of the selected words in a poem. The algorithm uses feature unification and constraint satisfaction to ensure congruency.

To finish the poem, the number of lines is either set as a constant, or determined by the bitstring. For each of the lines, the previously mentioned algorithms are executed on the bitstring. The indentation is a function of the next bitstring integer as well, working in similar ways as selection based on the bitstring happens in the previously described algorithms. The exact implementation of this function can be defined by the user of the EvoPoem system before letting it run. A new line can be started between any two words as well. Whether this happens should be determined by whether the next bitstring integer is higher than a certain user defined constant as well: the higher this constant, the lower the amount of premature line breaks.

The result of these algorithms is a poem parsed from a bitstring. The bitstring will always parse into the same poem, provided the grammar and lexicon do not chance, and is therefore deterministic.

Chapter 7 Results

For our implementation in Java, we used a simplified version of one of the example grammars provided by Jurafsky and Martin (2009, p. 462). This grammar defines a few options for noun phrases, verb phrases and prepositional phrases, and provides rules for how they can be combined to form grammatically correct sentences, as discussed in the previous section. We created a lexicon for all the word classes required by this grammar: nouns, verbs, articles, prepositions and pronouns. A dictionary could be used to provide a lexicon, but for this proof of concept, we made a custom selection of words for our lexicon. We selected some homonymous words, which could appear in multiple word classes. This was done to increase the chances of encountering double meaning, which is often used in the poetry we focus on. Our program generates random bitstrings, which are then parsed into a poem, using the algorithms described in the previous section. The features we used for this implementation are limited to person, number and tense. To distinguish between the forms of a noun, we used features for indicating whether an inflection is personal, distant or possessive. Another feature was added to nouns that indicated whether the word should be preceded by the 'a' article, or the 'an' article.

Figures 4 to 7 provide a small selection of some random four line poems that were generated using this algorithm.

the dives answered
damage
an answer dies
cured

Figure 4 Example poem

bandage
you cured
we alerted
they died

Figure 5 Example poem

they echoed
they ached
a claim acted
they ached

Figure 6 Example poem

cured
you cover
the cures count through you to the answers on the brushes
covered
we dived

Figure 7 Example poem

The reader should keep in mind that with these tests, the word selection is based on a completely random bitstring. No evolution has taken place with this algorithm so any form of meaning – or lack thereof – is purely coincidental and the presented poems are first generation candidates only; not the final result of a full machine learning enabled poetry generator. However, these examples clearly show the general form of the type of poetry we described before. The example 'poems' consist of a few short sentences and use indentation to change the perceived rhythm of the poem. Although the word choice makes it more difficult to perceive the poems as fully grammatically correct, the sentence structure is grammatical under the implemented grammar. In this case, the words the algorithm selected were a product of a first generation random bitstring, but should follow from the evolutionary process, not the bitstring parsing algorithm. Although the poems do not consist of sentences that would be used in a natural English conversation, the structure more than suffices to have them serve as a starting point for evolving poetry.

Chapter 8 Conclusion and Suggestions for Further Research

8.1 Conclusion

Our goal was to develop an algorithm that could use a simple grammar data structure that could hold complex grammars, to parse a bitstring deterministically to a small poem. We used the concept of context free grammars (CFGs) and feature mapping and we extended this concept to include features and unification, using constraint propegation to ensure congruency. This combines in a decision tree-like structure that is able to generate text. At each possible moment, a decision is made by parsing the next part of the bitstring. The result of our algorithm will always yield the same output (poem) for a given input (bitstring), provided that none of the parameters – such as the grammar and the lexicon – are changed.

In our implementation, we used a very simple grammar, with a very limited lexicon and only the most important features. This served well for testing purposes, as it allowed us to show our proof of concept algorithm satisfies all of our requirements. Although the small grammar and lexicon we provided allow for some variation in the poems that can be generated, our algorithm differs drastically from a fully developed natural language in the sense that the number of distinct poems that can be generated with this algorithm is far from infinite. With a more exhaustive grammar and lexicon, the search space for poetry would increase as well.

Implementing the evolutionary algorithm fell outside of the scope of our proof of concept. Because of this, the poems that were generated by our algorithm are, effectively, random, rather than the result of an intelligent process. This means the poems generated by our implementation of the algorithm should not be treated as an end result yet, in the same sense that the first generation of any genetic or evolutionary algorithm population should not be treated as the end result.

It should also be mentioned that the algorithm we developed entails a specific approach for generating grammatically correct texts and an internal representation that can be used for evolution. It is not, however, the only approach. As is often the case in programming, other algorithms can be developed that can meet all our requirements as well. Neither are our choices for the implementation of feature unification or for how to handle the constraint satisfaction search problem necessarily the only correct method.

Lastly, the poetry that is generated by our algorithm, or by an evolutionary algorithm that uses our algorithm to parse the individuals in a population, is the product of chance crossovers and mutations of previous poems. All elements of the poem, including meaning, are the product of combining previous successful combinations, and are not put in the poem with the specific intent to exhibit “poeticness”. This means the burden of assigning meaning is left with the reader and the poems generated with this algorithm cannot be classified as poems if one follows the definition penned by Manarung. Although Manargun’s definition seems to have become the standard, disregarding it allows for methods that differ from previous attempts. For our goals, the attributes of poeticness and rhythmically would have been too constraining, as they limit the way the final product can be shaped by how poems are rated by readers.

8.2 Limitations of the Current Study and Suggestions for Further Research

Implementing the evolutionary algorithm fell outside the scope of our proof of concept. The algorithm that we developed as our proof of concept, however, was designed with the intent to use it in combination with an evolutionary algorithm. We would be interested to see this aspect of the EvoPoem poetry generator implemented. We would suggest implementing a mutation operator and a cross-over operator in the bitstring object, where a mutation is the random flip of a bit and a cross-over is the exchange of a sub string of two bitstrings. The theories for evolutionary algorithms are well developed and the implementation of such an algorithm would not be very hard to realise, as we discussed before. We left this implementation out, however, because it would need to run for a very long time before yielding interesting results. We leave the decision of parameter values, such as mutation chance and population size, to the research team that will continue this project. As discussed in previous sections, we advise to use human readers to assess the quality of a generated poem and implement a fitness function that works on this user rating. Our suggestion would be building a website that shows two poems next to each other and asks the user to click on the poem he or she likes less than the other. The poem that was clicked is then replaced by another poem and the rating for the poems is updated in the database to indicate that the clicked poem is perceived as a worse poem than the one that was not touched. This method results in a relative rating between poems, rather than an absolute rating such as would be the result of using a scale. We believe this relative rating will handle user inconsistency better than an absolute rating. Due to the implementation of our algorithm, any other fitness function, such as automatic rating based on some features, will work with the algorithm just as well. When this system is designed and based on a user rating, an interesting research question would be that of what makes a poem a poem. Can a reader assign meaning to any text, regardless of intent, or is intent required to make a text be perceived as meaningful? An answer to this question would provide valuable insights into the human psyche.

Lastly, the data structure used to store the grammar and lexicon are complete in the sense that they can contain all necessary features and do not grow exponentially when they are extended, but they could hugely benefit from a more readable format. Research effort could be put in developing a data structure for this grammar and lexicon that can be read, understood, adapted and extended without extensive knowledge of programming and independent of the programming language in which the grammar is implemented.

References

Dempsey, I., O'Neill, M., & Brabazon, A. (2009). *Foundations in Grammatical Evolution for Dynamic Environments.* Springer.

Greene, E., Bodrumlu, T., & Knight, K. (2010). Automatic Analysis of Rhythmic Poetry with Application to Generation and Translation. *Proceedings of the 2010 conference on empirical methods in natural language processing.* Association for Computational Linguistics.

Jurafsky, D., & Martin, J. H. (2009). *Speech and Language Processing* (Second ed.). Prentice Hall.

Levy, R. (2001). A Computational Model of Poetic Creativity Width Neural Networks as Measure of Adaptive Fitness. *Proceedings of the ICCBR-01 Workshop on Creative Systems.*

Manurung, H. (1999). Chart Generation of Rhythm-Patterned Text. *Proceedings of the First International Workshop on Literature in Cognition and Computers.*

Manurung, H. (2003). *An Evolutionary Algorithm Approach to Poetry Generation.* University of Edinburgh.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms.* MIT Press.

Oliveira, H. G. (2009). *Automatic Generation of Poetry: An Overview.* Universidade de Coimbra.

Queneau, R. (1961). *Cent Mille Milliards de Poèmes.* Gallimard.

Russel, S., & Norvig, P. (2013). *Artificial Intelligence: A Modern Approach.* Pearson.

Toivanen, J. M., Toivonen, H., Valitutti, A., & Gross, O. (2012). Corpus-Based Generation of Content and Form in Poetry. *Proceedings of the Third International Conference on Computational Creativity*, (pp. 175-179).

Source

The source code of the proof of concept described in this thesis is available free of charge for reference and reuse from Bitbucket.com under the General Public License v3.0.

Via WWW https://bitbucket.org/automatedpoetry/evopoem-bitstring-parser