Automated Cut-Up Poem Generation from Large Corpora

Nathaniel K Smith

December 19, 2009

Abstract

In this paper I propose a new approach to computer poetry, presenting an algorithm that performs directed cut-up poem generation technique on a large corpus. I discuss in detail the technical implementation of the algorithm and its associated modules. To provide background and motivation, I present a survey of existing computer poetry generation techniques alongside an analysis of their ability to produce interesting results. To support the analysis, I offer a discussion of how the technical implementation of computer poetry generation affects the production of interesting results. I argue that my new algorithm fills both a technical and creative gap within the existing computer poetry landscape. Finally, opportunities for further research are discussed.

Contents

1 Introduction **2**
1.1 Background 3
1.1.1 Two Categories 3
1.1.2 Existing Projects 3
1.2 Analysis 5

2 Directed Cut-up Technique **6**
2.1 Explication of Algorithm 7
2.2 Technical Implementation 7
2.3 Corpus Acquisition 8
2.4 Corpus Processing 9
2.5 Rules System 10
2.5.1 Rule Class 10
2.5.2 Input to Rules Translation 11
2.5.3 Implemented Rules 11
2.5.4 Example 12

3 Results **12**
3.1 Selected Outputs 12
3.1.1 Couplet 13
3.1.2 Haiku 13

3.1.3 Limerick 14
3.1.4 Miscellaneous 14
3.2 Defense of Technique 15
3.2.1 Semantic Consistency 15
3.2.2 Variable Structure 15
3.2.3 Human Mediated, Not Human Authored 16
3.2.4 A Useful Seed 16
3.2.5 Conclusion 16

4 Further Research **17**
4.1 Missing Pieces 17
4.2 Applications 17

1 Introduction

It seems that most attempts made at 'good' computer poetry (a metric which is arguably even less measurable than 'good' poetry on its own) often fall prey to one of two problems: one, leaving too much to the computer, thus cutting out human spontanaeity and creativity or two, treating the computer as a bland random number generator and using an over-abundance of human-prepared content.

I posit that there is a way to synthesize these two extremes by reappropriating human content easily accessible via the Internet. Significant changes in both the amount of content stored online and the continuing ease of access to such content mean that there is a massive store of thoughts, emotions, humour, tragedy, and other elements of 'good' poetry just waiting to be used as inspiration for the computer-poet. To this end, I present an algorithm that 'cuts up' a given, large corpus (presumably harvested from the Internet) and produces a poem according to some desired output parameters.

In this paper, I discuss in detail my implementation of the algorithm and its associated modules. To provide background and motivation, a survey of existing computer poetry generation techniques is presented with an analysis of their ability to produce interesting results. To support my analysis, I offer a discussion of how the technical implementation of computer poetry generation affects the production of interesting results. I argue that the new, presented algorithm fills both a technical and creative gap within the existing

computer poetry landscape. Finally, opportunities for further research are discussed.

1.1 Background

1.1.1 Two Categories

My research into existing work in the field of computer poetry revealed two general categories of techniques: I refer to the first of thse as *templatized* and the second, *generative*. The former category relies on some input text and an algorithm to produce new works. The latter relies on a pre-defined grammatical (or semantic) template to produce work from scratch (in other words, without any input text).

To adapt the summation found in [Hartman1996], templatized computer poetry has this general flow: a human programmer creates some kind of initial structure (either semantic or grammatical) and uses it along with software to produce a work. A generative approach to computer poetry, on the other hand, provides a program that acts as a filter of sorts that takes some initial text–not necessarily written by the programmer designing the program–and converts it into a new work.

1.1.2 Existing Projects

Early attempts at computer poetry were templatized techniques. The earliest documented approach is Auto-Beatnik, which churned out randomly rendered poems using grammatical structures and a vocabulary of english words (nouns, verbs, pronouns, and so forth). It was not until 1984 that another serious attempt at poetry generation was publicised with the release of a work by the program RACTER called *The Policeman's Beard is Half-Constructed*. At 66 pages, *Policeman's Beard* was a much larger effort than the handful of short poems output by Auto-Beatnik. However, the generation process was essentially the same idea: the program called upon built-in grammatical templates and a large vocabulary of english words [Chamberlain1984]. Despite the 22 years between these two projects, they were both, at their core, automated versions of the popular game Mad Libs. In such a game, one player picks arbitrary nouns, verbs, and adverbs while another

player records them in the blanks of some written piece. Employing a computer in this process is trivial.

A more recent templatized technique is detailed in [Manurung et al.2000] and [Manurung2003]. This technique stochastically evolves a poem from a simple phrase or set of simple phrases into an interesting poem using a set of evaluation functions.

This approach is a wide ranging departure from RACTER. The technique, packaged in software named McGonagall, relies on the manipulation of grammatical information using a Lexicalized Tree Adjoining Grammar (LTAG). The LTAG is a derivation tree; in other words, it itself does not describe the phrase structure of a sentence. Instead, it describes operations performed on a set of phrases. Each node of the LTAG is an elementary tree describing some phrase structure and each edge labels the location at which a child node should be inserted into its parent node. These LTAG structures serve as a kind of template from which poems are derived. Potential solutions are evolved using the LTAGs and evaluated for things like meter, rhyme, and semantics.

An early and basic generative approach is the Travesty Engine discussed in [Hartman1996]. This technique accepts an input text and reorganizes it using a Markov chain (a random generative process in which future states are selected based only on the present state of the generation). Given some n, the Travesty Engine reassembles an input text t such that the result, s, contains all the same n length substrings in t. At $n = 1$ the program simply shuffles the letters of t; but at about $n = 9$, t begins to resemble a grammatically correct rearrangement of the phrases in s. Thus, for varying levels of n, new poetry could be produced from any input and this basic algorithm without the need for any kind of external lexicon.

A recent technique, and one similar to the technique I am proposing, uses Vector Space Modeling to produce Haiku using content found on blogs throughout the Internet [Wong and Chun2008]. This method produces modern (in other words, not strictly 5-7-5) Haiku poems using randomly accessed blog content.

Before any utilization of blog data, a keyword lexicon is first built using common Haiku terms. Using this lexicon, HTML output from a blog search engine is parsed and sentences

containing lexicon words are extracted and converted into fragments according to English grammatical rules. Finally, longer fragments are filtered out of the repository leaving only fragments of a length suitable for use in Haiku.

Three keywords are randomly picked from the lexicon and used to further narrow the sentence repository. The remaining fragments are then run through an analysis stage in which sentence pairs are picked based on a vector representation of their semantic similarity. Finally, the 'best' three lines are picked for the final Haiku, each one semantically involving one of the three random keywords originally picked.

Another approach similar to my technique is the case-based reasoning technique developed in [Gervas et al.2001]. This approach accepts prose sentences to be converted to poetry. This process is largely generative, making grammatical changes to the input sentence as well as word substitutions. The target output is specified by some existing stanza of the desired form (i.e., a haiku). An important part of the process, however, is a vocabulary of words that may be substituted into the original prose to achieve the desired poetic structure, making this approach a combination of templatized and generative techniques. A user must supply this vocabulary for each poem desired.

1.2 Analysis

While researching works extant in computer poetry it became clear that a question common to many projects was "why bother?" Templatized techniques tended to rely almost exclusively on the human programmers designing the poetry generation software–making it seem like the computer was little more than a set of dice. Even in [Manurung2003], where the templatized technique is taken to an extreme, the semantic scope of the produced work is still strictly limited to what humans provide in the LTAG collection. From an A.I. perspective, the work is incredibly impressive; but from a poetic standpoint, the process is still dominated by humans. Computer poetry such as this isn't adding much to the field that a human could not already to by hand.

The CBR approach suffers from similar problems despite being a partially generative

technique. A substantive part of the creativity of the system depends on the words chosen for the vocabulary used, meaning that a human must premeditate the content of produced poems. Similarly, a human user must select the poetic verses used to compare potential outputs to when generating a poem.

The question is easier to answer for generative techniques. The Travesty Engine, for example, produces new arrangements of pieces that humans may never think to produce. It can reveal new meaning in an existing piece through purely aleatoric means. Similarly, in the VSM Haiku generator, the produced poetry can describe the zeitgeist of the blogs it searches (something that even Google struggles to do). Clearly this poetry does something that a human couldn't do: revealing meaning hidden within existing human prose. I submit, however, that in the former example the rearrangement is too simple because it does not go far enough and its results become predictable after a fashion. In the latter example, the final Haiku bear too little resemblance to their source material to really articulate any kind of underlying meaning to, or really resonate, with the reader.

2 Directed Cut-up Technique

Thus, I present a new generative technique inspired by the work of the poet and author William Burroughs, who would manually cut into pieces his own writing and the writing of others to then rearrange the pieces into new narratives [Wikipedia2009]. My technique is an implementation of Burroughs's method, and exploits the diversity and large number of sentences found in large corpora–for example, Wikipedia, the online articles of a newspaper, or online public domain books–by scanning the sentences within them, profiling them, and then using them to populate a poem based on rules input by a human user. Thus, the technique *cuts up*, like Burroughs, some given corpus in order to produce a poetic result. I argue that, given a large and varied enough corpus, this technique produces interesting work, and present several features of the resulting pieces that support this.

2.1 Explication of Algorithm

The technique is driven by a short algorithm supported by several modules. It is presented in pseudo-code in figure 1. The algorithm, given some corpus C containing several thousand lines of text, stores each sentence of the corpus in a database with associated syllabic and phonetic information. Using any number of simple guidelines supplied by a user, it proceeds to construct a poem using the sentences stored in the database. Should the algorithm be unable to locate a sentence to satisfy some line of the desired poem–say, a sentence with 10 syllables containing the word *aleatoric* and rhyming with the word *trumpet*–it *weakens* the rule for that line, relaxing the criteria used in the database lookup to find a sentence that at least closely resembles one needed to meet the user's guidelines. Without this technique, termination is not guaranteed; however, every rule eventually weakens to the trivial rule, which is satisfied by any sentence.

```
Given some corpus C and database handle DB
L ← normalize(C)
insert(DB, profile(L))
i ← input()
R ← rulesParse(i) {a list of lists of Rule objects}
P ← ∅ {the output poem (a list of lines)}
for r ∈ R do
  S ← select(DB, ruleSetToQuery(r))
  if not S then
    while r′ ← weaken(r′) do
      S′ ← select(DB, ruleSetToQuery(r′))
      if S′ then
        S ← S′
        break
      end if
    end while
  end if
  s ← random(S)
  push(P, s)
end for
output(P)
```

Figure 1: A pseudo-code representation of the algorithm

2.2 Technical Implementation

The tools involved in the implementation of this algorithm are Perl 5.10[per2009] and SQLite3[sql2009]. I chose Perl due to its rich Lingua::EN repository which contains many

modules for parsing, splicing, and cleaning up English text. Used in the implementation of this algorithm were the CPAN modules Sentence[Yona2002], Splitter[James2005], Phoneme[Thurman2009], and Syllable[Fast1999] of the Lingua::EN namespace. I selected SQLite3 over any large RDBMS for its speed, in-RAM database feature and portability.

Despite initial concerns about the ability of an interpreted language to process enough text in a short amount time, the implementation ended up with favorable benchmarks. Producing 10 line poems with an AB rhyme scheme and 10 syllables per line from a SQLite3 database containing .4 million sentences consistently took between .2 and .4 seconds. Parsing a 1 million line corpus and inserting it into a SQLite3 database took 2.5 hours. The former benchmark indicates that the poem generation code will scale for larger corpora, but the latter shows that refactoring or rewriting will be necessary to support multimillion line corpora. All of this work was performed on a modern laptop running Ubuntu with a consumer grade dual-core CPU and 1 gigabyte of RAM.

It is important to note that there are numerous opportunities for parallelism in this project that could be included in future refactors. Corpus acquisition and poem production are both embarrassingly parallel operations; content can be downloaded asynchronously in the former and in the latter, a process can be spawned for each line of a poem-in-progress as no single line depends on any other line.

All code developed for this project is licensed under version 2.0 of the Artistic License and freely available at `http://github.com/nathanielksmith/weltanschauung`.

2.3 Corpus Acquisition

A key component of this approach is the presence of a large corpus. Years ago, before the Internet expanded to its current voluminous state, it was unlikely that any such corpus would be publicly and freely available. Now, one can consider the publicly accessible areas of the Internet as one huge corpus with smaller selections of Internet content as subsets of this corpus.

For initial testing and debugging, I manually developed and employed a digitized

version of a dozen short stories by Jorge Luis Borges.

For later testing and prototyping, I scraped with an automated script a selection of 3,700 books from Project Gutenberg[gut2009]. From these books, 1,000,000 lines of text (out of a total 7,000,000) were analyzed, producing 433,850 profiled sentences stored in a database for use.

I attempted a scrape of several thousand Wikipedia pages, but the content proved to be too diverse (in terms of encoding and structure) to be parsed without lots of additional overhead. The primary issues were UTF-8 encoding and the vagaries of sidebars, tables of contents, and headers.

2.4 Corpus Processing

For each sentence found in the input corpus, the following details are stored in a database:

1. line number
2. rhyme sound
3. syllable count
4. number of words
5. sentence text

The rhyme sound is determined through analyzing the phonetic representation of the last word of the sentence. Lingua::EN::Phoneme, a Perl module based on the CMU Pronouncing Dictionary, is used to break apart the word into its constituent phonemes. Its rhyme sound is defined to be the final vowel sound of the phoneme list followed by any number of consonant sounds. For instance, since the phonemes of the word *bear* is B, AO1, R, its rhyme sound is represented as ‘AO1R’ while *bears* would be represented as ‘AO1RS’.

Syllables are also counted using the Lingua::EN::Phoneme module. The syllable count of a word is defined to be the number of vowel sounds (in other words, phonemes with a string representation of length 3) found in its phonetic representation.

Unfortunately, relying on the CMU dictionary means that there are occasionally words which are not profiled in the listing. For these words, no rhyme sound is recorded. For syllable counting, the Perl module Lingua::EN::Syllable is fallen back to, which employs an

intelligent vowel count technique to produce syllable counts with about 80% accuracy[Fast1999].

The line number is used both as a primary key for the sentence's database representation as well as for determining line proximity.

2.5 Rules System

Once a corpus database is ready, user-inputted guidelines for the desired output are parsed into a list of lists of Rule objects that are capable of weakening themselves and producing SQL clauses to be used in a SELECT statement.

2.5.1 Rule Class

Though not included in the algorithm's original design, it became clear that the only way to properly organize the rule code was through an object-oriented approach. I designed a Rule class which keeps track of a 'weakness' level–the number of times the rule has been liberalized by the algorithm–and provides functions for weakening the rule returning a SQL clause that describes the rule in its current state.

Each rule object's get_clause function returns SQL suitable for appending to a SELECT statement's WHERE clause. For example, the unweakened SQL for a Rule::Syllable object initialized with an argument of 5 would be '(num_syllables = 5)'. Thus, the ruleSetToQuery needs only iterate over the rule objects for any line of the poem it's constructing and join each object's get_clause output with the string ' AND ' to result in a statement like those in figures 2 and 3.

Once weakened, a rule object's get_clause function returns SQL modified to be more general. Instead of giving weights to any kind of rule, each rule object in the rule set for a given line of a poem is equally likely to be randomly weakened. To illustrate the weakening process, figure 2 shows a trace of the SQL generated for the first line of a sample 10-line AB scheme poem using a small (600 sentence) corpus.

As you can see in figure 2 the algorithm struggled to find good candidate sentences

```
(end_rhyme_sound=\'AE1N\')AND(num_syllables = 10)
(end_rhyme_sound=\'AE1N\')AND(num_syllables BETWEEN 10-1 AND 10+1)
(end_rhyme_sound=\'AE1N\')AND(num_syllables BETWEEN 10-2 AND 10+2)
(end_rhyme_sound=\'AE2N\')AND(num_syllables BETWEEN 10-2 AND 10+2)
(end_rhyme_sound=\'AE0N\')AND(num_syllables BETWEEN 10-2 AND 10+2)
(end_rhyme_sound=\'AE0N\')AND(num_syllables BETWEEN 10-3 AND 10+3)
(1) AND (num_syllables BETWEEN 10-3 AND 10+3)
```

Figure 2: Trace of SQL during weakening for one line

given the size of the corpus; this was intentional to show how rules weaken. Note that the trivial rule manifests itself as simple '(1)', a clause sufficient to select any sentence.

2.5.2 Input to Rules Translation

In the current implementation, output guidelines are specified through command line arguments to a main driver program. With the exception of the length rule, each rule specific on the command line describes a scheme–in other words, a layout for each line of the poem.

For example, given the string ABABCC, the program will first pick a random sound for each unique letter in the scheme and later use them to select sentences with that rhyme sound for each line's corresponding rhyme label modulo the number of lines. Thus, a 12 line poem with the scheme ABABCC would produce a poem with ABABCCABABCC rhymes.

Similarly, one expresses syllable schemes using a comma separated list of syllable counts. To specify a Haiku, one would input a length of 3 and a syllable scheme of "5,7,5".

Once each input rule is parsed, a list of Rule objects for each line of the poem is created.

2.5.3 Implemented Rules

At publication, the following rules are implemented:

1. Rhyme: Match the rhyme sound of the last word of a sentence
2. Syllable: Match the syllable count of a sentence
3. Keyword: Match lines that either contain a given word (exact) or are near such a line (fuzzy)

2.5.4 Example

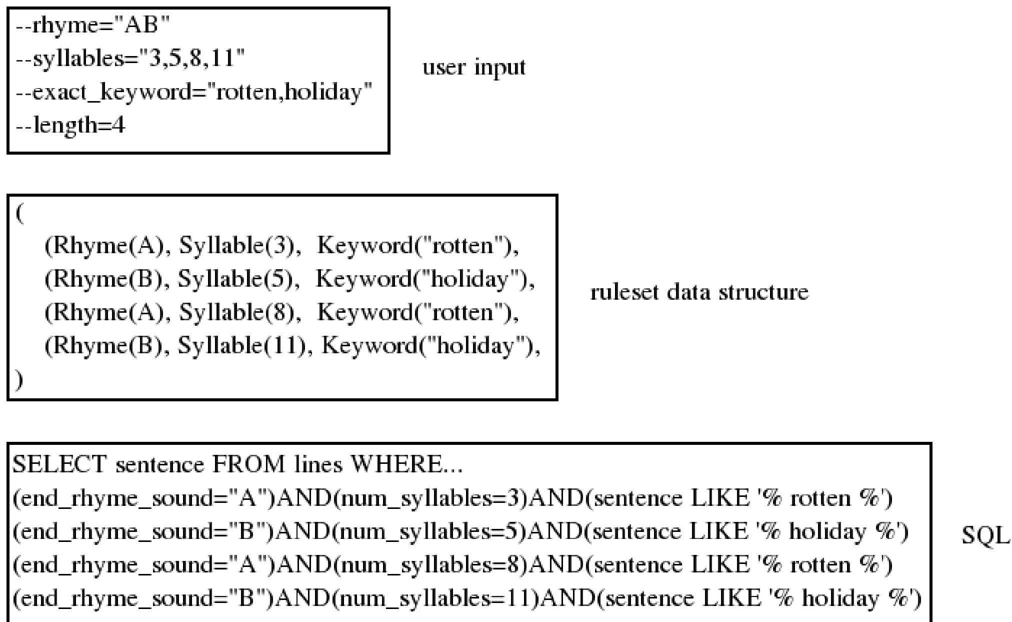

Figure 3: Rules, from user input to SQL

3 Results

3.1 Selected Outputs

The following examples were selected from several runs of the algorithm against the 433,850 sentence corpus from Project Gutenberg. All poems received minimal editing; only stray newline characters were removed.

3.1.1 Couplet

These samples were generated with the input:

```
--rhyme="A" --length="2" --syllables="10"
```

"You havent seen him, have you?
"People seldom do.

"Let none falter who thinks he is right."
They saw a truly marvelous sight.

Blumpo dances a breakdown for joy.
"Oh, you tell it," grumbled the fat boy.

"Will I?" Answered the small youth.
He guessed a great deal of the truth.

"We cant limit procreation by law.
In short there was no end to what he saw.

But the affair was growing serious.
Her prim phrasing turned to earnestness.

Figure 4: Various couplets

Note the accidental semantic consistency in these pieces. Not only do the lines often sound good together, it is easy for a reader to immediately find meaning and coherence in the lines.

3.1.2 Haiku

Classic Haiku poems are 3 lines long with a 5-7-5 syllable pattern. Due to the imprecise syllable counting the algorithm must often fall back on, the Haiku presented here are not as strictly defined. However, they have the same look and feel. They were generated with these guidelines:

```
--syllables="5,7,5" --length="3"
```

Harrys son nodded.
Three columns and two arches.
GLORY MAY NOT LAST.

Read All About It!"
She caught disconnected words.
FRANCOIS PASSES BY

Jack picked it up.
"It sounds complicated."
The woods are on fire!

Then I began:
"Dont worry about me."
IN THE HOSPITAL

Alexander said.
"Boys, whose animal is this?"
A Crazy Mans Doings.

The spring was coming.
"What sort of looking men?"
You will share in their work.

Figure 5: Various Haiku

In these Haiku especially, one finds that many of the better poems are either funny, owing to jarring combinations of strange phrases, or hauntingly poignant. Of particular note is the first haiku listed in figure 5. One can imagine a family staring out over the ruins of the Acropolis and pondering the inevitable demise of imperial glory.

3.1.3 Limerick

Limericks are typically obscene pieces belonging largely to folk traditions. The algorithm was capable of producing limericks that traded obscenity for absurdity. The rules used to generate the following were:

```
--rhyme="AABBA" --length="5" --syllables="10,10,6,6,10"
```

Cake--What variety gives to life.
He had led such a terrible life!
And truly it was so.
At least I think so.
It was the best period of his life.

The cook did not require anything more....
I cant think where Ive met you before."
"My scheme succeeded.
Sorry we intruded."
I knew the writing; there was no more.

Figure 6: Various limericks

Given the line constraints it was difficult to produced topical limericks that did not weaken too much. But limericks of contrary absurdity were easy to come by and seemed to fit the spirit of limericks anyway.

3.1.4 Miscellaneous

These pieces were created with a variety of inputs. Some are topically oriented, others purely random. They were selected for their weight and coherence.

You had charge of the funeral arrangements.
Gabriel, why did you set your heart on me?"
"Hes mostly there this time o day."
"Its my lungs Im worried about," Mary said.
The bodily heat falls very rapidly.
"Hes mostly there this time o day."
There was no tribute but their tears.
You had charge of the funeral arrangements.
[Sidenote: Result of the contest.]
He did not want to let Renovales go.
But the contest irritated the king.
That husky young boy was her son.
"Did they tell you, Mariano?
She must stay at home and work for others.

Although the cargo was taken out,
    it was after it had been in the water
    more than one half months.
Updated editions will replace the previous one--
    the old editions will be renamed.
The soldiers were ordered not to
    allow him to place on the vessel

Everybody was marching away.
You must be on your guard.
What shocks you so?
Where are your eyes?"

"Thats a good one for you, Upton.
The old woman laughed in derision.
They meddle with the royal jurisdiction.
You have arranged no consultation?"

The first official report sent
    by Governer Francisco de Sande
    to the home government
    is dated June 7, 1576.
What troops can be away out here?
War had brought about many changes.

Figure 7: Various freeform works

Without syllable constraints, pieces end up with very long lines, as shown by the indented line continuations in figure 7. Such lines, though, are often poetic unto themselves and can work well within even a short piece. The first piece listed in figure 7 was made with the fuzzy keyword "death" and features a good mix of related and unrelated lines, sufficiently creating a tone for the whole piece.

3.2 Defense of Technique

3.2.1 Semantic Consistency

By using one or more topics as rules in the input to the software one can topically flavor the output without restricting it. Sentences relevant to the selected topic will be used to produce the final piece. The reader should be able to get a sense of what each produced poem is "about." It is important to note that this semantic consistency does not undermine juxtaposition, an important aspect of computer poetry [Hartman1996]. The output will still contain the often jarring combination of seemingly unrelated phrases despite their semantic consistency.

3.2.2 Variable Structure

Poems produced by my technique can scale from completely arbitrary, random productions to very narrowly defined poetic structures (i.e. sonnets, Haiku, or limericks). The rules engine system allows for great flexibility and is a key component of this technique. Its

success, however, relies on the theory that large enough corpora will contain sentences diverse enough to satisfy any input rules.

3.2.3 Human Mediated, Not Human Authored

A crucial aspect of my approach is that, while all of the content used to produce works is written by humans, the meaning of the content is subverted and altered by the reappropriation process to create something new in which the humans and computers involved both play integral roles in the poetic production. I refer to the technique as being *human mediated* since humans provide the media–prose sentences–while the computer acts as the artist. In this case, however, the computer is more like an artist of reappropriation, such as a DJ, collage artist, or cut-up author.

3.2.4 A Useful Seed

As noted in [Hartman1996], computer poetry can be a useful starting point for human poets. This approach produces material that is easily reworked into a more coherent piece by a discerning human artist. Often, merely changing just one or two words in an output poem can make it match a particular scheme or achieve some other poetic end. The software was demonstrated for human poets in a local reading group and each one found the results compelling and inspiring for her or his own work.

3.2.5 Conclusion

This approach is a versatile one that employs the computer in a necessary and novel way to produce interesting poetry. The rules system is sufficiently 'hands-off' such that a human merely suggests the final poetic structure of a piece while the computer performs a task that is humanly impossible. The 'Why bother?' of this approach is answered thus: new, creative work is discovered and enjoyed with the necessary help of a computer.

4 Further Research

4.1 Missing Pieces

The most glaring lack in this implementation is the absence of a scansion rule. Without this rule, most major poetic forms cannot be approximated. A scansion engine[Hartman] licensed under the GPL exists in the Python language and could be adapted to work with the Perl source of this project. Similarly, the current definition of rhyme used in this project is naïve, only accounting for the final vowel sound of a word. Future iterations of this work will account for the many different kinds of rhymes possible in a given poem (e.g., syllabic, imperfect, or slant).

Further, I need to develop a faster way of acquiring and profiling large corpora. Refactoring and parallelizing parts of the existing code seem to be the most expedient way to accomplish this. Parallelization can also improve the production speed of longer poems with several rules per line.

Finally, the resulting pieces need to be cleaned up in some automated way–stray punctuation should be corrected, as should erroneous characters that survived the profiling process (like newlines). Apostrophes, too, need to be intelligently handled as for now they are simply stripped out. As a part of this project a Lingua::EN::Unapostrophe module was designed which would strip possessive apostrophes but expand apostrophes representing word contractions–from don't to do not, for example–but at time of publication the design has not been implemented.

4.2 Applications

The quality of this approach's output depends largely on its corpus. One possible "interesting" corpus is a mixture of the complete works of two poets, potentially radically different, to create surprising combinations. Combining Plath and Whitman, for example, could illustrate the often oscillating moods of self-fulfillment and despair. Similarly, news content from sources with different political agendas (e.g., the Huffington Post, generally

regarded as liberal, and the Drudge Report, generally regarded as conservative) could be combined to reveal excesses of hyperbole on both sides of a political debate.

A more subtle application of this software is a kind of fuzzy data mining for sociological research. The works produced by corpora containing every status update of every friend of social network users from different socioeconomic strata could elucidate differences in interests, literacy levels, and biases between classes.

Finally, the Internet itself is a corpus. Works produced from it would seem infinitely diverse and could describe the zeitgeist of the Internet's billions of users. In order to accomplish this, data stores like Google's would be necessary, but it is technically feasible given the right resources.

References

[per2009] (2009). Perl 5.10. `http://www.perl.org/`. [Software].

[gut2009] (2009). Project gutenberg. `http://www.gutenberg.org/wiki/Main_Page`. [Website].

[sql2009] (2009). Sqlite3. `http://www.sqlite.org/`. [Software].

[Chamberlain1984] Chamberlain, W. (1984). *The Policeman's Beard is Half-Constructed: Computer Prose and Poetry.* Warner Software/Books.

[Fast1999] Fast, G. (1999). Lingua::en::syllable. `http://search.cpan.org/~gregfast/Lingua-EN-Syllable-0.251/Syllable.pm`. [Software].

[Gervas et al.2001] Gervas, P., Sistemas, D., and Programacin, I. (2001). Automatic generation of poetry using a cbr approach. In *In CAEPIA - TTIA 01 Actas Volumen I. CAEPIA*.

[Hartman] Hartman, C. O. Scandroid. [Software].

[Hartman1996] Hartman, C. O. (1996). *Virtual Muse: Experiments in Computer Poetry*. Wesleyan University Press.

[James2005] James, D. (2005). Lingua::en::splitter. `http://search.cpan.org/~splice/Lingua-EN-Segmenter-0.1/lib/Lingua/EN/Splitter.pm`. [Software].

[Manurung2003] Manurung, H. M. (2003). An evolutionary algorithm approach to poetry generation.

[Manurung et al.2000] Manurung, H. M., Ritchie, G., and Thompson, H. (2000). Towards a computational model of poetry generation. In *In Proceedings of AISB Symposium on Creative and Cultural Aspects and Applications of AI and Cognitive Science*, pages 79–86.

[Thurman2009] Thurman, T. (2009). Lingua::en::phoneme. `http://search.cpan.org/~marnanel/Lingua-EN-Phoneme-0.01/lib/Lingua/EN/Phoneme.pm`. [Software].

[Wikipedia2009] Wikipedia (2009). Cut-up technique — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Cut-up_technique&oldid=319936739`. [Online; accessed 21-October-2009].

[Wong and Chun2008] Wong, M. T. and Chun, A. H. W. (2008). Automatic haiku generation using vsm. In *Proceedings of The 7th WSEAS International Conference on Applied Computer and Applied Computational Science*, pages 318–323.

[Yona2002] Yona, S. (2002). Lingua::en::sentence. `http://search.cpan.org/~shlomoy/Lingua-EN-Sentence-0.25/`. [Software].