German University in Cairo
Faculty of Media Engineering and Technology
Computer Science Department

Automatic Poetry Generation Using CHR

Masters Thesis

Author:	Alia El Bolock
Supervisor:	Prof. Slim Abdennadher
Co-supervisor:	Prof. Thom Frühwirth
Submission Date:	November 24, 2014

This is to certify that:

(i) The thesis comprises only my original work toward the Masters Degree.

(ii) Due acknowledgement has been made in the text to all other material used.

Alia El Bolock
November 24, 2014

Acknowledgments

To start off, some people, without which this work could not have been completed, need to be acknowledged. I would like to show my gratitude to

- My parents and fiancé for their constant love, support and encouragement.
- My friends and family for always being there, even from the distance.
- My friends here in Ulm, for the fun times, the mutual help and for providing an outlet to bounce off ideas.
- Amira, for always being there when needed. For the constant support, patience and help in the theoretical, practical and personal matters. Finally, for answering all our questions and concerns without hesitation and encouraging us.
- Prof. Thom Frühwirth for giving insight and guidance when needed. For steering the project in the right direction but also giving me the freedom to shape the project on my own.
- Professors Thom Frühwirth and Slim Abdennadher for providing me with this topic.
- Last but not least, I would like to thank god, for finishing my bachelor project and thesis.

Abstract. Poetry is one of the most interesting and complex natural language generation (NLG) systems, because a text needs to simultaneously satisfy three properties, to be considered a poem; namely poeticness (poetic structure), grammaticality (grammatical structure and syntax) and meaningfulness (semantic content). In this thesis, we discuss the development and implementation of an autonomous system, capable of generating unique yet meaningful poetry, that harnesses the advantages of Constraint Handling Rules (CHR). This is realized through the implementation of a reasoner, which generates poems, that satisfy poeticness, grammaticality and meaningfulness, based on a customized lexicon. In the proposed system, a poem is generated by incrementally selecting its words, through a step-wise pruning of the lexicon by the reasoner. This is done based on the constraints, that represent poeticness, grammaticality and meaningfulness. The developed approach proves, that CHR can be used to develop a hybrid system capable of generating good poetry comparable to, and matching that of humans.

Contents

Contents **IX**

1 Introduction **1**

2 Preliminaries **5**
2.1 Constraint Handling Rules 5
2.1.1 Syntax 5
2.1.2 Example 7
2.2 Poetry 8
2.2.1 Definition 8
2.2.2 The Three Properties 8
2.2.3 Characteristics and features 9
2.3 Resources 11
2.3.1 CMU Dictionary 11
2.3.2 Alternate 12 Dicts Package 13
2.3.3 Wordnet 13

3 Approaches and Techniques for Poetry Generation **17**
3.1 Grammar- and Template-Based Approaches 17
3.1.1 The Poetry Creator 17
3.1.2 RACTER 17
3.1.3 The ALAMO Group 18
3.1.4 Others 18
3.2 Generate and Test Approaches 18
3.2.1 Manurung's chart system 18
3.2.2 WASP 19
3.2.3 ASPID 19
3.2.4 Tra-la-Lyrics 19
3.3 Evolutionary Approaches 20
3.3.1 MCGONAGALL 21
3.3.2 POEVOLVE 21
3.4 Case-Based Reasoning Approaches 22
3.4.1 COLIBRI 22
3.4.2 ASPERA 23

3.5 Constraint Programming and Corpus-based Approaches 24
3.5.1 Constraint- and Corpus-Based Poetry Generation 24
3.5.2 Full-FACE 25
3.6 Other Approaches 26
3.6.1 Stochastic Language Modelling 26
3.6.2 POS-Tag Based 26
3.6.3 Vector Space Model 27

4 System Architecture **29**
4.1 Grammaticality 30
4.1.1 Grammar Pattern List 31
4.1.2 Grammar Correction 32
4.2 Poeticness 33
4.2.1 Basic Features 33
4.2.2 Figures of Speech 33
4.2.3 Form 34
4.3 Meaningfulness 35
4.3.1 Theme-based Lexica 35
4.3.2 Poem Actors' Restriction 35

5 Lexicon **37**
5.1 Design 37
5.2 Generation 39
5.2.1 Java 39
5.2.2 CHR 41
5.2.3 Additional Information 41

6 Implementation of the Reasoner **45**
6.1 Initialization and Termination 45
6.2 Basic Grammaticality and Poeticness 46
6.3 Additional Grammatical Constraints 48
6.4 Additional Poeticness Constraints 49
6.5 Coherence 50
6.6 Choices 51

7 Semantics **53**
7.1 Concept 53
7.2 Semantic Network Generation 53
7.3 Integration 54

8 Evaluation **57**
8.1 Quantitative Evaluation 57
8.1.1 Survey 57
8.1.2 Results and Analysis 60
8.2 Comparison 66

8.3 Web Application 67

9 Conclusion and Future Work **69**
9.1 Conclusion 69
9.2 Future Work 69

References **73**

List of Figures

2.1 The phoneme set used by the CMU Dictionary 12

4.1 The pruning steps of the selection process of each word in the poem . . . 29

8.1 Results of Question 1 61
8.2 Results of Question 2 61
8.3 Results of Question 3 62
8.4 Results of Question 4 63
8.5 Results of Question 5 64
8.6 Results of Question 6 65

Chapter 1

Introduction

Computational Creativity is a research field, at the cross section of Artificial Intelligence, Computational Linguistics and Cognitive Science, that aims at automatically generating creative artifacts such as musical pieces, stories, visual art and poetry. Having a system capable of producing or simulating human-level creativity helps give more insight into human creativity and could define it in an algorithmic way. This could also allow for programs to augment the human creativity to possibly unexplored horizons. One of the most expressive and creative uses of language is poetry, as it offers freedom, in terms of writing rules, and is highly based on human interpretation. Also unlike other types of text, a poem's value depends on both its content and its form. As a result, poetry is one of the most interesting and complex natural language generation (NLG) systems. For a system to be capable of generating acceptable poetry, we first need to define the notion of poetry. In Manurung (2004), Manurung postulates that a text needs to simultaneously satisfy three properties, to be considered a poem, namely poeticness (poetic structure), grammaticality (grammatical structure and syntax) and meaningfulness (semantic content).

The aim of this thesis and the conducted research, is to develop an autonomous system, capable of generating unique yet meaningful poetry, that harnesses the advantages of Constraint Handling Rules (CHR). Throughout the thesis, the hypothesis, that a CHR-based system is be capable of generating poems comparable to human written ones, is tested and proven. To achieve this, various poetry generation approaches were investigated, before developing our own hybrid approach to achieve automatic poetry generation. The thesis, discusses the implementation of a poetry generation system, based on the developed approach. This is realized through the implementation of a reasoner, which generates poems, that satisfy poeticness, grammaticality and meaningfulness, based on a customized lexicon. In the proposed system, a poem is generated by incrementally selecting its words, through a step-wise pruning of the lexicon by the reasoner. This is done based on the constraints, that represent poeticness, grammaticality and meaningfulness.

The main contributions of this thesis are:

1. A hybrid poetry generation approach and system, that produce texts satisfying grammaticality (syntax pattern), poeticness (rhythm and rhyme) and meanigfulness, simultaneously.

2. A single, large lexicon that provides all the necessary information, allowing for the creation of a large variety of poems and increasing the odds of finding an optimal solution. The lexicon does not follow the conventional Part-of-Speech tags (POS-tags) but an extended version thereof with further word categorizations that lead to more semantic meaningfulness and syntactic correctness with less constraints and computations.

3. Instead of relying on complex computational methods, the system follows an intuitive rule based algorithm enabled by CHR's intuitive multi-headed rule representation, that allows the manipulation of the constraint store without further specifications; e.g. the rhythm is fulfilled using simple pattern matching. The algorithm enables the generation of the whole poem, without being iterations, and is thus more efficient than other systems. The use of Prolog's lists coupled with CHR's constraints maintains efficiency while still taking advantage of the needed benefits of using constraints.

4. The developed system does not rely on any corpora, which cause other systems that are corpus-based to produce results that are still similar to their base text, even with their requirements for novelty, as they all perform some sort of "cutting and pasting" on these base texts. Instead, our system can optionally use training texts to extracts their morphological info, providing a list of POS-tags and nothing more.

This thesis is organized as follows:

Chapter 2 Preliminaries: the preliminary knowledge needed for understanding the work is explained.

Chapter 3 Poetry Generation: an overview of the state of the art and the various approaches and techniques used in automatic poetry generation, is given.

Chapter 5 System architecture: an outline of the system architecture is presented and the approach used for the poetry generation is explained.

Chapter 4 Lexicon: the design and generation of the lexicon is discussed.

Chapter 6 Implementation of the Reasoner: the various stages of the implementation of the reasoner are explained.

Chapter 7 Semantics: the extension of the implemented system with semantics is discussed.

Chapter 8 Evaluation: the quantitative results of a survey are presented and analyzed. The poems generated are compared to those of other existing ones.

Chapter 9 Conclusion and Future Work: the conclusions and results of the work are presented and the possible future work is discussed.

Chapter 2

Preliminaries

In the following a brief overview of the used poetry terms and linguistic resources will be given, and the programming language Constraint Handling Rules will be introduced. A basic knowledge of linguistics is assumed.

2.1 Constraint Handling Rules

Constraint Handling Rules (CHR), is a high-level, constraint-based, declarative logic programming language, invented by Prof. Thom Frühwirth in 1991. CHR adapts the basic concepts of mathematical logic representation and is thus highly and easily applicable to various problems. CHR is a committed-choice, single-assignment language, with multi-headed rules and conditional rule application through guards. A CHR program consists of CHR rules that add and remove the user-defined constraints to and from the global constraint store. Having simplification, propagation and simpagation (a mixture of the afore mentioned rules) as the only operators that can deal with constraints, CHR is well suited for representing constraint-based problems and solving them straightforwardly. However, CHR has evolved into a general-purpose programming language. CHR is usually used as an extension for a host language; most commonly Prolog, although it can be ported to various different environments. The properties of CHR enable the user to design anytime, online, confluent and concurrent algorithms, depending on the semantics used. Frühwirth (1998) is an excellent reference for more detailed explanations of CHR, its properties and advanced examples. In the following however, a brief overview will be given.

2.1.1 Syntax

Any CHR rule has the following attributes:

- Name r_i: This is an optional attribute to identify a given rule

- Head H^k, H^r: Any rule must have one or many heads. These heads may be kept or removed. Unification with the head(s), triggers the application of the rule. The head can only consist of CHR constraints; i.e. user defined constraints.
- Guard G: The optional guard, is a condition for the application of the rule. After the heads are unified, the rule is only fired if the guard is true. The guard can only consist of built-in constraints.
- Operator <=> or ==>: The operator is responsible for deciding which operation should be done with the constraints, upon firing of the rules. More to the operators in the sections below.
- Body B: The body of the rule comprises of the constraints to be added to the constraint store upon application of the rule. The body can consist of both CHR and built-in constraints.

2.1.1.1 Simplification Rules

Simplification rules look as follows:

$$r_s \ @ \ H^r \iff G \mid B$$

Simplification rules remove the set of constraints in the head and replace them with a set of "simpler"- for the problem- constraints from the body. A simple example of a simplification rule is

```
hungry <=> eat.
```

Where `hungry/0` and `eat/0` are user defined constraints. This rules translates to: If one is hungry and eats, he is no longer hungry so the `hungry` constraint can be removed.

2.1.1.2 Propagation Rules

Propagation rules look as follows:

$$r_p \ @ \ H^k \implies G \mid B$$

Propagation rules add new constraints from the body without removing the preexisting ones. They simply use the knowledge one already has, to deduce further information. An example for such a rule is

```
happy ==> smile.
```

Here, `happy/0` and `smile/0` are constraints defined by the user. This means that one is happy, he will smile as a result, but smiling will not remove his happiness.

2.1.1.3 Simpagation Rules

Simpagation rules look as follows:

$$r_{sp} \ @ \ H^k \setminus H^r \Longleftrightarrow G \mid B$$

This type of rules merges between the two other types, mentioned above. While is introduces new constraints to the constraint store, part of the head constraints are kept in the constraints store while other superfluous ones are removed. This can be explained on the example of the one rule minimum algorithm in **?**

```
min(N) \ min(M) <=> N=<M | true.
```

This algorithm works through comparing two user defined `min/1` constraints from the constraint store and removing the one with the larger number, as this number can not possibly be the global minimum. After exhaustive application, the only remaining `min` constraint will be the actual minimum of the input set.

2.1.2 Example

The syntax and semantics of CHR will be explained further by the example of the partial order relation $\leq$, `leq(X,Y)`, which holds if the value of variable `X` is less than or equal that of `Y`. The CHR program for defining the `leq` relation consists of four rules:

```
reflexivity @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y) , leq(Y,X) <=> X=Y.
idempotence @ leq(X,Y)\ leq(X,Y) <=> true.
transitivity @ leq(X,Y) , leq(Y,Z) ==> leq(X,Z).
```

The simplification rule responsible for representing the reflexivity of the `leq` relation consists of one head, that simplifies the fact, that `X` is less than or equal `X`, to true. This is done by removing the `leq(X,X)` constraint from the constraint store. The anti-symmetry property is again described using a simplification rule with two heads, where two symmetric `leq` constraints mean that the two variables are of equal value. Thus, the two are removed and the equality relation between them is added to the constraint store and handled by the host language, namely Prolog. The idempotence rule is responsible for removing redundant copies from the constraint store by keeping the constraints before the '\' and removing the ones following it. Finally, The transitivity rule is a propagation of a new leq constraint, that can be implied from the already existing ones, while the constraints in the head are kept in the constraint store. The execution of a CHR program is triggered by an input query , where the constraints of the query keep firing the rules by matching the heads until a fix point is reached. For a sample query

of `leq(X,Y), leq(Y,Z), leq(Z,X)`, the transitivity rule will add `leq(X,Z)`, which will trigger the anti-symmetry rule to remove `leq(Z,X),leq(X,Z)` and add `X=Z`. The unification of `X` with `Z` makes the anti-symmetry rule applicable on the first two constraints of the query, and results in the addition of `X=Y`. At this point there are no more constraints in the store to match the rule heads. Thus, no more rules can be fired and the program will terminate and output the answer `X=Y,X=Z`.

Throughout the thesis, please note the distinction between CHR constraints that represent actual objects and the semantic constraints enforced on poems.

2.2 Poetry

2.2.1 Definition

Finding a concrete definition of poetry, is a difficult task, because poetry has various different forms and genres and highly depends on subjective taste and interpretation. The Merriam-Webster English Dictionary defines a poem as:

"a piece of writing that usually has figurative language and that is written in separate lines that often have a repeated rhythm and sometimes rhyme"

This shows that the content and form of a poem are strongly interacting and equally important for its value. This is what differentiates between poetry and prose texts in the first place. This becomes clear, for example, when we think about translating a poem compared to another text. The poem, will be much harder to translate and will probably lose its beauty while maintaining its meaning; the other text will remain the same. This is so, because in poetry various phonetic and figurative devices interact to provide the feeling and meaning of a poem. This is supported by LEVIN (1962), where it is stated that:

"... in poetry the form of the discourse and its meaning are fused into a higher unity ... form in fact embraces and penetrates message in a way that constitutes a deeper and more substantial meaning than either abstract message or separable ornament."

2.2.2 The Three Properties

As no definition of poetry can be followed to define how valid poetry should look like, we follow the representation of Manurung (2004). It states that a text has to satisfy three properties to be considered a poem:

- **Meaningfulness** This means that a poem must convey a certain message, that has a meaning under some interpretation, given a specific knowledge base. It is clear that this property should be true of any text, not only poems.

- **Grammaticality** This means that a poem must follow the linguistic rules of a certain language, which are defined by a grammar and a lexicon. Again, this feature needs to be satisfied by any given text, and not only poetic ones. the only difference here is, that poems are less constrained in terms of grammar, and thus some lenience can be applied when considering the grammatical rules. However, caution needs to be applied when doing so, because too much deviation from grammaticality, could turn the text from poetic to not understandable.

- Poeticness This means that a poem must have poetic features, that differentiate it from other texts. These features are figurative as well as phonetic and form dependent. Rhythm, rhyme, metaphors, repetitions etc. are all key features that make poems special.

So, a poem is a text that fulfills the properties of meaningfulness, grammaticality and poeticness.

2.2.3 Characteristics and features

In this subsection, we introduce the different characteristics and features of poetry and define the basic terminology. All the used definitions are collected or exactly taken from Manurung (2004); Ltd (2014); Foundation (2014); Campbell (2014).

2.2.3.1 Basic Terminology

First we will give some definitions of recurring terms in poetry:

- **Stanza**: A group of lines that is separated from others in a poem. Stanzas are used to shift between moods, time, action or thoughts.

- **Verse**: "A line in a poem or a line of poetry". Can also we used to refer to formal poetry, in general.

- **Phoneme**: "A distinct unit of sound in language to distinguish between words. For example p, b, d, and t in the words pad, pat, bad, and bat."

- **Syllable**: "A single unit of a whole speech sound; i.e. a vowel preceded by consonants (from zero up to three) and followed by consonants (from zero to four). For example tag, gross, strings."

2.2.3.2 Rhythm

In poetry, the rhythm of the language plays an important role and is clearly defined and followed. In the following the different rhythm-related terms and concepts will be defined:

- **Stress**: "A stressed syllable, is one that is more emphasized than the others". In English, the words in themselves have a definition, of which syllables are stressed and which are not. However, the position in a sentence, the semantics and the chosen metre, can influence the stress of a word.
- **Metre**: "The metre is the rhythmic pattern of a verse, represented by stressed and unstressed syllables."
- **Syllabic Metre**: "The most common metre in the English poetry. Here, the verses are divided into units, known as feet, which have a certain number of syllables. The metrical feet are defined in terms of the length of the syllable and the location of the stressed syllable." Some of the most used feet are:

  1. Trochee: 2 syllables, stressed - unstressed; e.g. incest
  2. Iamb: 2 syllables, unstressed - stressed; e.g. inject
  3. Dactyl: 3 syllables, stressed - unstressed - unstressed; e.g. terrible
  4. Amphibrach: 3 syllables, unstressed- stressed - unstressed; e.g. incumbent
  5. Anapest: 3 syllables, unstressed - unstressed - stressed; e.g. interrupt

  The feet are then used in a certain sequence for each verse. The most popular is the iambic pentameter, which is a sequence of five consecutive iambs.

2.2.3.3 Rhyme and other phonemic patterns

As discussed before, other than the rhyme, that governs the flow of the whole poem, poetry is known for using many phonemic patterns, such as rhymes. In the following, the most common ones will be defined:

- **Rhyme** is "the similarity of the ending sound of two words. Two words rhyme, if their final stressed vowel and the sounds following it, are matching." There are two types of rhyme:

  1. In a masculine rhyme only one syllable is rhyming; e.g. rot and not.
  2. In a feminine rhyme two or more syllables are included in the rhyme; e.g. lighting and fighting.

- **Alliteration** is "the repetition of identical consonant sounds, usually at the beginning of a word; e.g. she sells seashells."

- **Anaphora** is "the repetition of the same word or phrase at the beginning of a line, in the whole poem for part of it."
- **Assonance** is "the repetition of identical vowel sounds in different words, that appear close to each other; e.g. deep green sea."
- **Consonance** is "the similarity in sound between two words; e.g. bed, bad."
- **Dissonance** is "the disruption of harmonic sounds or rhythms; e.g. cacophony."
- **Enjambment** is "the continuation of a sentence from one verse to the next."

2.2.3.4 Figurative Language

The beauty of poetry lies in the freedom of interpretation. The figurative language is responsible for allowing the poet to become more creative using imagery and the poem reader to use his or her imagination and interpretation. There are various different types of figurative language: the symbolic imagery type and the phonemic type. In the following, we will give the definition of the most widely used figures of speech and stylistic devices:

- **Metaphor** is a direct comparison, but without the explicit use of comparative words; e.g. like, as.
- **Oxymoron** "joins contradictory words to create an effect; e.g. deafening silence."
- **Personification** is "the description of a material item, a concept or non-human, as though it were a person."

2.3 Resources

2.3.1 CMU Dictionary

The Pronunciation Dictionary for North American English by the Carnegie Mellon University University (2014) is machine-readable dictionary with over 125000 entries of words and their pronunciations. The dictionary's format makes it suitable for speech recognition and synthesis system, because it maps words to their phonemic pronunciation. The phoneme set used in the dictionary consists of 39 phonemes, shown in Figure 2.1. The vowel phonemes, also carry a lexical stress, to show the stressing of the syllable represented by the vowel. In the dictionary, there is a distinction between three different vowel stresses.

- No stress: represented by the number 0 attached to the vowel
- Primary stress: represented by the number 1 attached to the vowel
- Secondary stress: represented by the number 2 attached to the vowel

Phoneme	Example	Translation
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
IH	it	IH T
IY	eat	IY T
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
OW	oat	OW T
OY	toy	T OY
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
UH	hood	HH UH D
UW	two	T UW
V	vee	V IY
W	we	W IY
Y	yield	Y IY L D
Z	zee	Z IY
ZH	seizure	S IY ZH ER

Figure 2.1: The phoneme set used by the CMU Dictionary

2.3.2 Alternate 12 Dicts Package

The Unofficial Alternate 12 Dicts Package Beale (2014), consists of various files. For the purposes of thesis, we will only describe the file relevant to our implementation; the 2of12id.txt file. This file consists of a huge number of words, with all their inflections. The file differentiates between the basic word types: noun, verb, adjective/adverb, conjunction/preposition, interjection, pronoun, spoken contraction. For example, for a base noun word, its plural is also stated, if it exists. Similarly, this holds for the verbs and adjectives and their different inflections.

2.3.3 Wordnet

Wordnet University (2010); Witzig (2003) is a "lexical reference system" and database, which is also available in Prolog format. In this thesis, we use the Prolog version of Wordnet, as Prolog is the host language we use for CHR in this work. Wordnet is based on the concept of synonym sets, called synsets. A synset is a group words, which are semantically connected. Every word has a synset ID, and words of a synset have the same ID. Words that have various meaning, will belong to different synsets and thus will have multiple IDs. In the Prolog database, this means that some words will have multiple entries with different synset IDs. The different word relations contained in Wordnet, are represented using operators. In Prolog, each operator has a file (with the name of the operator), that contains all the Prolog clauses of the word pairs, linked by a said operator. The operators describe semantic as well as lexical relations. The operators used for the incorporation of semantics in our poetry generation system are listed below. Their definitions, is taken from the official Wordnet documentation University (2010).

1. `s(synset_id,w_num,'word',ss_type,sense_number,tag_count).` "A `s` operator is present for every word sense in WordNet. In `wn_s.pl` , `w_num` specifies the word number for word in the synset."

2. `sk(synset_id,w_num,'sense_key').` "A `sk` operator is present for every word sense in WordNet. This gives the WordNet sense key for each word sense."

3. `hyp(synset_id,synset_id).` "The `hyp` operator specifies that the second synset is a hypernym of the first synset. This relation holds for nouns and verbs. The reflexive operator, hyponym, implies that the first synset is a hyponym of the second synset."

4. `ins(synset_id,synset_id).` "The `ins` operator specifies that the first synset is an instance of the second synset. This relation holds for nouns. The reflexive operator, `has_instance`, implies that the second synset is an instance of the first synset."

5. `ent(synset_id,synset_id).` "The `ent` operator specifies that the second synset is an entailment of first synset. This relation only holds for verbs."

6. `sim(synset_id,synset_id).` "The `sim` operator specifies that the second synset is similar in meaning to the first synset. This means that the second synset is a satellite the first synset, which is the cluster head. This relation only holds for adjective synsets contained in adjective clusters."

7. `mm(synset_id,synset_id).` "The `mm` operator specifies that the second synset is a member meronym of the first synset. This relation only holds for nouns. The reflexive operator, member holonym, can be implied."

8. `ms(synset_id,synset_id).` "The `ms` operator specifies that the second synset is a substance meronym of the first synset. This relation only holds for nouns. The reflexive operator, substance holonym, can be implied."

9. `mp(synset_id,synset_id).` "The `mp` operator specifies that the second synset is a part meronym of the first synset. This relation only holds for nouns. The reflexive operator, part holonym, can be implied."

10. `der(synset_id,synset_id).` "The `der` operator specifies that there exists a reflexive lexical morphosemantic relation between the first and second synset terms representing derivational morphology."

11. `cls(synset_id,w_num,synset_id,w_num,class_type).` "The `cls` operator specifies that the first synset has been classified as a member of the class represented by the second synset. Either of the `w_num`'s can be 0, reflecting that the pointer is semantic in the original WordNet database."

12. `cs(synset_id,synset_id).` "The `cs` operator specifies that the second synset is a cause of the first synset. This relation only holds for verbs."

13. `vgp(synset_id,w_num,synset_id,w_num).` "The `vgp` operator specifies verb synsets that are similar in meaning and should be grouped together when displayed in response to a grouped synset search."

14. `at(synset_id,synset_id).` "The `at` operator defines the attribute relation between noun and adjective synset pairs in which the adjective is a value of the noun. For each pair, both relations are listed (ie. each `synset_id` is both a source and target)."

15. `ant(synset_id,w_num,synset_id,w_num).` "The `ant` operator specifies antonymous word s. This is a lexical relation that holds for all syntactic categories. For each antonymous pair, both relations are listed (ie. each `synset_id,w_num` pair is both a source and target word.)"

16. `sa(synset_id,w_num,synset_id,w_num).` "The `sa` operator specifies that additional information about the first word can be obtained by seeing the second word. This operator is only defined for verbs and adjectives. There is no reflexive relation (i.e. it cannot be inferred that the additional information about the second word can be obtained from the first word)."

17. `ppl(synset_id,w_num,synset_id,w_num)`. "The `ppl` operator specifies that the adjective first word is a participle of the verb second word. The reflexive operator can be implied."

18. `per(synset_id,w_num,synset_id,w_num)`. "The `per` operator specifies two different relations based on the parts of speech involved. If the first word is in an adjective synset, that word pertains to either the noun or adjective second word. If the first word is in an adverb synset, that word is derived from the adjective second word."

Chapter 3

Approaches and Techniques for Poetry Generation

Automatic poetry generation started developing as a research field in the late nineties when the first promising systems started to emerge. Since then various systems using a large range of approaches have been appearing. In order to investigate new advantageous methods for poetry generation, the state of the art in this growing field needs to be reviewed. In the following advantages of each system will be marked in italics while disadvantages will usually be stated explicitly.

3.1 Grammar- and Template-Based Approaches

Systems where an incomplete poetry template is filled with words from a certain dictionary to suit a set of defined syntactic and/or rhythmic constraints. Notable examples of template-based poetry generation are:

3.1.1 The Poetry Creator

According to Oliveira (2009), the Poetry Creator is a simple poetry generation system, that fills predefined poem templates with words that describe a certain subject, a synonym for that subject and a title for the poem, all of which are input by the user.

3.1.2 RACTER

RACTER Manurung et al. (2012) is assumed to employ grammar-based generation as it performs verb conjugation and noun declension. It also gives the impression of thematic continuity by employing a heuristic of reusing some lexical elements. The resulting texts are thus similar to understandable sentences, which enabled the publication of the poems produced by RACTER e.g. "The Policeman's Beard is Half Constructed" consists solely of RACTER generated poetry.

3.1.3 The ALAMO Group

There are various french poetry generation programs on the ALAMO group website. The description of some of these programs is given on the website e.g. the program Rimbaudelaires Rubaud J and P (2000); Gervás (2002) uses existing Rimbaud sonnets as a starting template and then replaces the nouns, verbs and adjectives by words appearing in Baudelaire's poetry, following strong syntactic and rhythmic constraints.

3.1.4 Others

Other notable template-based poetry generation systems include:

- ELUAR: In addition to the template-based generation approach, EULAR simulates the appearance of coherence and poeticness by utilizing a heuristic e.g. assigning ad-hoc emotional categories like love, nature and philosophy.
- ADAM
- PROSE Manurung et al. (2012): Some of the poems generated by PROSE has been published.
- ALFRED the Agent (Donald n.d.)
- Masterman's haiku generator (Boden 1990)

3.2 Generate and Test Approaches

Generate and test systems produce random word sequences following constraints, that satisfy some poetic formal requirements e.g metric or semantic constraints. In the following some of the most important systems in this category are discussed.

3.2.1 Manurung's chart system

In Manurung (1999) Manurung introduced using a chart system to generate *syntactically correct texts* that conform to a *rhythmic pattern*. Usually charts are used for parsing but reversing a chart parser provided Manurung with a chart generator that translates logical forms to strings and is thus used to generate natural language strings matching a certain stress pattern. The *input semantics* are described by first order predicates that logically represent sentences. During the generation phase, the stress pattern of the result of a new rule is checked against the target stress pattern before adding the rule to the chart. Stress patterns are represented as lists of weak and strong syllables which are obtained from a pronunciation dictionary. The system maintains syntactic, semantic, and rhythmic well-formedness at every step at the cost of being computationally very expensive and not very flexible, as it can only generate a perfect result or none at all. This system was used within the more advanced poetry generation system McGonnagall Manurung (2004) which will be discussed in detail later.

3.2.2 WASP

WASP Gervás (2000a) is a forward reasoning rule-based system which is considered one of the first serious attempts on automatic poetry generation. The system actually consists of multiple programs, each implementing a different construction heuristics acquired from formal metric constraints aiming at creating a poem based on prior poems given to the system and a set of words and reference verse patterns both provided by the user. Finally WASP either outputs a set of free verses or a set of verses following a certain strophic form; both in Spanish. Despite the fact that the produced poetry conforms to the rules of *formal metrics*, it fairs poorly from a linguistic point of view as it made little sense.

3.2.3 ASPID

The ASPID system Gervás (2000b); Gervás et al. (2001) for Spanish poetry generation requires the user to propose a message for the output poem and then uses certain algorithms to select a working set of words from an initial vocabulary based on the similarity between the user-defined message and a corpus of validated verses. Candidate words to be added are chosen based on their satisfaction of *strict metric and rhyming constraints*. The system defines priorities among the whole vocabulary based on said similarity calculations. Words added to the poem thus follow a ranking, where words with lower priority are only selected if no higher priority word can be selected. This procedure provides *better search times* and allowed the expansion of the vocabulary while utilising stricter formal constraints. Successful termination of the program is however not ensured above a certain threshold of the vocabulary size and the number of constraints. Also the system provides poor syntactic and semantic results.

3.2.4 Tra-la-Lyrics

Tra-la-Lyrics Oliveira et al. (2007b,a) is a system that generates Portuguese lyrics based on the rhythm of a song melody that is input by the user. By using the pattern of strong and weak melody beats as the rhythmic pattern, the process of creating song lyrics is almost identical to that of poetry generation. Following the generate and test approach, the system produces grammatical sentences and scores them according to the constraints of the derived metric pattern. To reach its goal Tra-la-Lyrics follows three strategies, taking into consideration the features of song lyrics e.g. rhyme and repetition in addition to the metric constraints:

1. Random words & rhyme: Only rhythmic constraints are applied when choosing words. This strategy however enables setting up the probabilities of reusing words and of rhyme locations

2. Words following sentence templates & rhyme: Not only rhythmic but syntactical constraints, given by sentence templates, are applied when choosing words, with

syntactical constraints taking precedence over rhythmic ones with backtracking in case of unsatisfied constraints. Also here the setup of the probabilities of reusing words and using words with given roots is supported in addition to trying to end grammatical and musical sentences in the same beat.

3. Grammar & rhyme: The words are chosen following sentence templates and then evaluated against musical sentences. Following multiple generations, the sentence that fits the target rhythm pattern best is chosen.

Strategies two and three however do not always ensure the presence of rhyme while the first strategy does not account for grammar rules.

3.3 Evolutionary Approaches

Evolutionary computing follows techniques based on concepts of biological evolution e.g. natural selection and genetic inheritance and are well suited for modeling the process of poetry generation as it is similar to the process human authors follow while writing poems.

Michalewicz (1994) defines an evolutionary algorithm as a multi-point stochastic search algorithm, meaning a heuristic search that explores multiple points in the search space simultaneously and avoids getting trapped in local maxima like other hill-climbing algorithms through stochastically navigating the search space. Evolutionary algorithms maintain a population of individuals, each representing a possible solution to the given problem, over some time t. The algorithm consists of five main steps:

1. Initialization: Construct a new population representing a set of starting points for exploring the search space, with the points ideally being spread evenly across the space.

2. Evaluation: Evaluate each possible solution to measure its fitness for use.

3. Selection: Form a new population by stochastically selecting individuals from the older population, while preferring fitter individuals.

4. Evolution: Transform some of the members of the new populations through genetic operators resulting in new solutions.

5. Repeat: Steps 2 to 4 are repeated until:

   (a) a certain number of iterations has elapsed

   (b) a certain fitness score is reached

   (c) the algorithm has converged to a near-optimal solution

3.3.1 MCGONAGALL

McGonnagall is an evolutionary system introduced by Manurung in his thesis Manurung (2004) that represents the process of poetry generation as a state space search problem using stochastic hill-climbing search, where each state represents a potential text with all its representations and a move from semantics to phonetics can happen at any representation level. The system generates *metrically constrained* poems based on a *given topic* using a grammar-driven formulation Manurung et al. (2000b), Manurung et al. (2000a). The generation process consists of two stages:

1. Evaluation phase: A group of individuals is formed based on initial information, target semantics and target phonetics. The individuals are scored based on different characteristics like surface form, phonetic pattern and semantics.

2. Evolution phase: A subset of individual with the highest scores is selected for reproduction and is thus mutated to hopefully produce better versions of the poem.

evaluation and evolution. The system reaches a goal state when the produced poem satisfies all three properties poetic texts must fulfil according to Manurung, namely *meaningfulness*, *grammaticality*, and *poeticness*. McGonnagall realizes grammaticality through the use of lexicalized tree-adjoining grammar (LTAG) while meaningfulness and poeticness are achieved by maximizing the evaluation functions that measure the isomorphism degree between the phenotypic features of a candidate solution and the target semantics and target metre (represented the same way as described in 3.2.1). The system models the creative process of real people due to the strong interaction between content and form in the generation process but has the disadvantage of being a knowledge-intensive approach. Although McGonnagall is capable of finding optimal solutions for moderately-sized target semantics and metre pattern separately, it has difficulties with simultaneously considering both evaluation functions Manurung et al. (2012).

3.3.2 POEVOLVE

Levy's work Levy (2001) takes the actual process of human poetry writing as a reference for creating an evolutionary computational model of poetry generation and a prototype instance of said model, namely POEVOLVE which aims at creating limericks. Following Levy's model a poetry generation system consists of:

- One or more generator modules responsible for creating the initial population of candidate poems and modifying them in the following generation instances.

- A workspace where the population resides

- A lexicon

- A conceptual knowledge base
- A syntactical knowledge base

In POEVOLVE the initial population is formed from a set of words along with their phonetic and stress information. First words that follow the appropriate *rhyme patterns* are selected before choosing words to fill the rest of the line based on their *stress information*. The evolution step of the genetic algorithm followed by POEVOLVE is realized by mutation and crossover operators that modify the words in the limericks while a neural network trained on human judgment performs the evaluation step. The main drawback of the system is its lack of consideration of syntax and semantics.

3.4 Case-Based Reasoning Approaches

Another popular approach for poetry generation is case-based reasoning, where existing poems are retrieved and then adapted based on the required content and a target message input by the user. Case-based reasoning poetry generation systems are forward reasoning rule-based systems that, when given an output message and a rough specification of the poem type, perform the following tasks:

1. select appropriate metre and stanza
2. generate draft poem
3. request validation or modification from the user
4. update database with the validated verse

3.4.1 COLIBRI

COLIBRI Díaz-Agudo et al. (2002) is a Spanish poetry generation system, very similar to ASPERA 3.4.2. COLIBRI stores the various cases in flexible representation following a Description Logic System and incorporates an application-dependent ontology (CBROnto), responsible for improving the system's inference power in addition to the representation and use of more explicit and general knowledge. The system takes a list of keywords representing the meaning and a specification of a certain strophic form as input. The generation algorithm then proceeds as follows:

1. A case following the appropriate strophic form is retrieved from a corpus of existing poems
2. The user-defined keywords are replaced in the text while conserving the *syntactic well-formedness*

3. The result is revised by replacing words to ensure conformity with the *metre and rhyme* specified by the strophic form

Although the text aims at conveying a certain user-defined message (albeit being trivially achieved), the final step of the algorithm might destroy this intended meaning in order to satisfy the constraints of the strophic form Manurung et al. (2012).

3.4.2 ASPERA

ASPERA Gervás (2000b), an evolution of the WASP system 3.2.2, employs a case-based reasoning approach to produce an improved version of the construction strategies developed in WASP. ASPERA requires the user to input a set of prose sentences describing the intended message. The system then generates Spanish poetry based on the given input text through composition of the poetic fragments, best matching the input, that are retrieved from the case base of existing poems. The fragments are combined through applying additional *metrical rules* to produce a final poem with a certain *relation to the input sentence*. The system ensures that the words in the verses are combined following the *syntax* of the language and that the result makes sense according to the word semantics. Depending on the chosen stanza, *rhyming* constraints are also maintained at line endings. The generation process followed in ASPERA requires the user to input the length and degree of formality of the required poem to extract the most appropriate strophic form from the knowledge base. The vocabulary selection is also manipulated by the user through a choice for the mood and setting of the poem. ASPERA does not model the complexity of natural language to achieve its results, as it relies on engineering solutions based on the input and its case base instead of a rich lexicon, syntax and semantics rules. The creation of each line in the main algorithm follows the four steps of case based reasoning systems:

1. Retrieve step: For each sentence in the user-intended message, retrieve a verse from the corpus of verse examples
2. Reuse step: Construct a draft of the line based on the POS-tag of the chosen verse
3. Revise step: The user validates or revises the draft
4. Retain step: Analyse and store the validated poems, to have their information available for reuse in further generations

The system selects words only based on their syntactic category and ranking with respect to the user-defined meaning, case description or case solution. Thus it lacks strong poeticness in that it does not consider metric information in the word choice, as it is not included in the lexicon to start with. The author suggests improving the word selection process in addition to incorporating semantic information through a knowledge rich ontology to improve the results. A later version of the system introduced in (Gervás, 2013) follows a similar approach that basically relies on choosing an existing poem, replacing some words in it according to some constraints and splitting the original lines using various methods.

3.5 Constraint Programming and Corpus-based Approaches

The found systems belonging to this category utilize constraint programming coupled with the reliance on corpora to extract certain information. Constraint programming on its own is popular for creative computation in the field of music Boenn et al. (2010); Sneyers and De Schreye (2010). In the field of poetry however, although constraints are often utilized in different forms in the process of automatic poetry generation, very few systems where found that are implemented using a constraint solver.

3.5.1 Constraint- and Corpus-Based Poetry Generation

Toivanen et al. (2012); Toivanen et al. (2013) introduce a poetry generation system that consists of two sub-components, described separately in two papers; the first sub-component is a conceptual space specifier that follows a corpus-based approach for fulfilling grammaticality and coherence (what poems can be like) and the second is a conceptual space explorer that uses constraint programming techniques for achieving poeticness features and actually producing the poems accordingly. This approach *separates* the content and form of a poem from the actual process of creating such poems

The specifier component is built according to the principles introduced in Toivanen et al. (2012), which on its own is a system capable of producing Finnish poetry with the help of *two separate plain text corpora*; a background corpus for mining *lexical associations* and a grammar corpus that provides *grammatical and structural patterns*.

The poetry generation algorithm used proceeds as follows:

1. Give or randomly choose a one word topic for the new poem
2. Extract words associated with said topic from the background graph representing common-sense associations between words extracted from the back ground corpus
3. Randomly select a text piece from the grammar corpus
4. Morphologically analyse the words in the text for their POS, case, tense etc.
5. Independently substitute the words in the text with the words associated with the topic while maintaining the original morphological forms and reverting to the original word if the morphological form cannot be matched
6. After going through all words, measure the the poem novelty based on the percentage of replaced words and only output if it is sufficiently different else retry the process for a different starting text

This system still did not cover the poeticness property of poetry which will be realized by the explorer component. The role of the specifier component is to use the input from the

user, alongside the corpora to generate the input to be passed to the explorer component; namely a large number of mutually dependent word choices for different positions in the poem and their dependencies.

The explorer component, described in Toivanen et al. (2013), then takes this input and produces the output poem by using a constraint programming approach based on searching for optimal solutions over an implicit representation of the conceptual space. The system solely relies on expressing all various aspects of poetry as interacting constraints and thus utilizes a constraint solver to find solutions.

The explorer component comprises of a constraint solver and a static library of constraints (provided by the system designers) . The data, that triggers the constraints or potentially creates new ones, provided by the specifier component is:

- Poem Skeleton: the number of lines with their respective number of words
- Candidates: a list of potential words for each position in the skeleton
- Form Requirements: possible requirements on the poem form, e.g. *rhyming structure, rhythm*
- Syntax and Content Requirements: possible requirements on the syntax and content of the poem, e.g. word inter-dependencies

The user can manipulate these parameters by providing the specifier component with his or her preferences.

The dynamic specifications provided by the specifier component alongside the constraint library form a constraint satisfaction and also optimization problem. The authors use answer set programming (ASP) Gelfond and Lifschitz (1988); Niemelä (1999); Simons et al. (2002) as the constraint programming paradigm for finding the solutions to the problem, i.e. to generate the poems. ASP is a data-centric paradigm where the input data (predicates) provided by the specifier component, express the constraint satisfaction problem. The actual poetry generation process is expressed as rule-based constraints , which are used to infer additional knowledge on the input data and to apply constraints on the possible solutions until the optimal one is found.

3.5.2 Full-FACE

The Full-FACE poetry generation system, Colton et al. (2012), uses templates to generates poems following a corpus-based approach according to given constraints on meter, stress, sentiment, word frequency and word similarity. The system analyses newspaper articles to determine a mood for the day, which is used to determine an article to base the poem on and a template for the poem.

The algorithm of poetry generation consists of four steps:

1. Retrieval: Similes (very short phrases) are mined from the internet, according to both sentiment and evidence.
2. Multiplication: Objects, aspects, description words, or any combination thereof ate substituted to create variations for each simile
3. Combination: Similes, their variations and phrases extracted from newspaper articles are combined according to template given by the user. Templates define which words must exactly match, the POS tags of words and how words can be combined to produce compound phrases.
4. Instantiation: Random phrases are chosen from and elaborate set to fill the fields of a user-given template.

The system *creates an aesthetic* based on lyricism, flamboyancy, sentiment and relevance to the article and looks for an instantiation that maximizes said aesthetic. It also provides a commentary for the generation process, in order to add value to the creative act. Constraints are used to shape only some aspects of the generation process of the described poetry generator.

3.6 Other Approaches

Some other approaches towards poetry generation have been tried and will be discussed briefly:

3.6.1 Stochastic Language Modelling

Markov-chains (n-grams) can be used to model some syntactic and semantic characteristics if language in a clear and simple way; n-grams can for example model the probability of certain words appearing next to each other. This can be used to produce simple poetry but at the expense of sentence and poem structures.

Ray Kurtzweil Cybernetic Poet, Kurzweil (2001), is a poetry generation system that generates *well-formed rhythmic* texts based on a statistical language model trained on a corpus of existing poems. The use of n-grams provides for *coherence within each verse* but not globally.

3.6.2 POS-Tag Based

et al. (2013) introduces a POS-tag based Basque poetry generation system that extracts POS-tag sequences from verse corpora and calculates the probability of each sequence. Three different experiments have been tried out for the generation process: Based on a strophe chosen from the corpora:

1. Replace each word according to its POS-tag and suffixes
2. Replace each noun and adjective with an equally inflected word
3. Only replace nouns with semantically related ones, using the Basque WordNet

The third experiment produced the best results with respect to *coherence* and *correctness*, however the system does not account for any poeticness features, like rhyme or metre.

3.6.3 Vector Space Model

Wong and Chun (2008) present an approach for generating modern haikus based on text found in blogs using a Vector Space Model (VSM). The system requires a keyword lexicon, consisting of 50 words commonly used for writing haikus, and a line repository, containing sentence fragments from blogs. The haiku generation algorithm proceeds as follows:

1. Choose three keywords from the lexicon to form the general picture
2. Search the line repository for fragments using these keywords
3. Using a weighing scheme to determine the importance of a word in a sentence, extract two keywords from each of the fragments
4. Describe the semantic relation between sentence pairs using vectors and create a query for each possible pair using a keyword from each sentence.
5. Assign the result of the query in Yahoo! to the corresponding element in each vector
6. Chose the sentences with the most semantically related vectors' pair for the final haiku

Summary

The variation in the ranges of output quality is clearly noticeable as some approaches only attempt to satisfy metrical or grammatical features while others focus more on syntax, semantic and coherence or even all three properties of a poem defined in Manurung (2004).

Because of the creative nature of poetry it is difficult to define an objective evaluation method of the quality of the resulting poems. Basic Turing-like tests (to be discussed later) and trials of publishing resulting poetry are often used to evaluate poetry quality. Also some work is done to try an define an objective criteria for assessing the creativity of the poems produced by a certain system Oliveira (2009).

Although computer generated poetry is still far from matching the quality of human written poetry, it is clear, that the rich and diverse research and implementation attempts towards this endeavor, provide many interesting and promising results, that show the potential of the field.

Chapter 4

System Architecture

After having mapped out the preliminaries and the state of the art of automatic poetry generation, in this chapter the approach used for the poetry generation will be discussed.

An overview of the whole system architecture will be given, before going into the poetry generation components in more detail. In the introduced approach, the whole poem is regarded as an empty grid, where the reasoner replaces each grid element with a word. A poem is generated by filling the grid, one element at a time. Each final word is selected from its set of candidate words, which is initially the whole lexicon. The selection process of each word can be regarded as a step-wise pruning of the whole lexicon by the reasoner, based on various constraints, as shown in Figure 4.1. The first constraint to be taken into consideration is the word type, where the POS-tag pruner selects only the words matching the current required word type, from the lexicon. The required word types are all stored in a grammar pattern list, that contains the POS-tag of all the words that should appear in the poem, in the correct order. All the important information, contained in the grammar list, is extracted from the lexicon, as it is explicitly designed to provide all the necessary grammatical details. The generation of the lexicon and grammar list will be discussed in more detail, later. After achieving a list of all the words, that

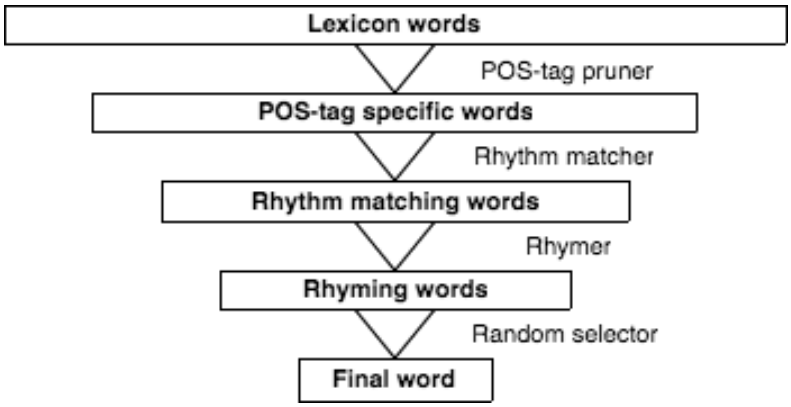

Figure 4.1: The pruning steps of the selection process of each word in the poem

match the grammar pattern, the list is narrowed down further, by the Rhythm matcher to keep only words corresponding to the current stress pattern. The stress pattern list contains the information about the rhythm of the poem and thus, ensures that all chosen words satisfy the stress pattern. The two pruning stages employed to this point, are sufficient to create an artifact that can be considered as English text, because it satisfies the basic grammar rules. In turn, this text can also be considered a poem, as it satisfies the most basic poem property, namely the rhythm. The quality of the generated poems, can however still be improved. In addition to the further grammaticality and poeticness elements, that need to be considered, the complex property of meaningfulness needs to be satisfied. Most of the further grammar rules are handled in advance in the initial grammar list, to ensure that the words are chosen correctly from the beginning. However, some grammar rules can only be applied after, all or some, words of the sentence have already been chosen. Another property that is familiar in poems, is the fact that verse endings follow a certain rhyme scheme. This is achieved by the rhymer, which enforces an extra pruning step to the candidate lists of words appearing at line endings. Finally the random selector selects a random word from the pruned candidate list, and adds it to the poem. As some words need to be compared to already chosen words, as well as the candidate list, CHR is very well suited for this type of constraint solving, due to its multi-headed nature, as will be shown later. Meaningfulness is achieved in this system, in the sense of poem coherence. It is ensured that each generated poem follows one basic theme by dividing the basic lexicon into various smaller theme-based lexica, and allowing the user to specify the theme of the poem to be generated. To achieve more coherence the system minimizes the number of acting subjects and objects in the poem, which will be discussed later. Because of the figurative and interpretative nature of poetry, texts generated using the above outlined approach sound meaningful and coherent. Also, because of the randomness of the word selection process, given the satisfaction of the basic constraints, the generated text has the beauty of enabling the readers to allow their imagination to run freely. Given a poem, each reader will deduce a different message he or she believes the poem conveys, which is the actual point of poetry and creative works, in general. The architecture of the reasoner enables the addition of any further constraints the user wishes to enforce on the generated poems, as it will simply require the addition on an extra pruning step to the candidate list.

The poetry generation process thus consists of two components, namely the lexicon and the reasoner, which will each be explained in further details in the following chapters.

As previously mentioned, a poem was defined to be a text that simultaneously satisfies three properties; poeticness, grammaticality and meaningfulness. In the following, we will discuss the approach chosen to realize each of the three properties using CHR.

4.1 Grammaticality

First we will discuss how the grammaticality property was achieved, meaning how we ensure that the generated text is grammatically correct. Instead of generating a text

and then checking if it is grammatically correct, it is better to generate grammatically correct text, to start with. To do that, we first need to define the correct English sentence structure. There are two options for defining the sentence structure of the poem: either by providing a grammar for the English language to specify what kind of sentences is accepted by it, or by explicitly giving a template containing the required POS-tags for each position in a sentence. The former option would be more autonomous, but requires handling the ambiguity of the English language and would allow for a smaller subset of the English language to ensure termination. It can also be investigated by implementing the grammar in CHR Grammars Christiansen (2004), the CHR counterpart of DCGs for Prolog. For the purposes of this thesis, the latter option was chosen, to investigate how grammaticality could be realized through the underspecification of constraints. The approach used depends on a list containing the target grammar pattern of the whole poem, namely the correct POS-tag sequence of the words that will appear in the poem.

4.1.1 Grammar Pattern List

The grammar patter list is the guide the reasoner uses to choose words from the lexicon and insert them in the poem. The first instance of grammaticality is thus ensured by always selecting words from the lexicon that match the current required word type, i.e. POS-tag. As long as the sentence structure represented in the grammar pattern list is a correct one the generated sentence should be grammatically correct. The reasoner goes through the list and selects all the words that match the current POS-tag, to then prune them further to match the remaining constraints. The grammaticality is only ensure through the POS-tag matching with the word types in the grammar pattern list and without any application of further constraints. The grammar pattern list, which is the core grammaticality measure, can be generated in many ways.

4.1.1.1 Manual Specification

The first possibility is the obvious one, of manually deciding on the required grammar pattern list. The user could specify the exact pattern of the required poem. This way he could manipulate the flow of the poem, by enforcing a certain grammatical structure. For example if the user, wants a poem that describes a certain concept, he would specify that the POS-tag of the first word should be a concept, followed by a verb and an object, then another verb and object, etc. However, this possibility is trivial and tedious for the user to fully specify the whole grammatical pattern of the poem.

4.1.1.2 Extraction from Corpora

The second option is to extract the grammar pattern of the poem, from corpus of existing poems. This can be done in two ways. The ifrst one is by collecting a corpus of poems for different poets or poetry types and then learning the different grammar patterns. This

way the user can specify the theme or poet style he wants and the grammar pattern matching the user's request will be used. The second possibility is to search the internet for poems matching the user's query and they extracting the grammar pattern from it. While matching the grammar pattern to that of an existing poem, the ambiguity of grammar parsing has to be handled. A naive instance of parsing to extract the grammar pattern from existing poetry has been implemented throughout this work. However, this approach was abandoned, to be enhanced later, as the goal of this work was to examine the possibility of poetry generation without relying on any external resources, to test the power of CHR for natural language generation. Also the approach of grammar pattern extraction from corpora has been investigated in numerous works so far, as shown in 3, and thus is was decided to pursue a novel approach.

4.1.1.3 Automatic Generation

The final option is the automatic generation of the grammar pattern list, which is the actual approach pursued in this work. The grammar pattern list is initially empty and the system just fills it with a certain number of nouns, to specify how many main sentences should appear in the final poem. Any noun is automatically preceded with an article, initially 'a'. The article will be removed or modified as needed, later. After any noun a verb should appear. Optionally, two nouns can first be combined with a conjunction before following them by a verb. Another optional feature, is the conjunction of two sentences; at this point two nouns and two verbs. Sentences that consist of only a noun and verb would produce text of very crude nature. This is where the lexicon design comes in. As will be shown in 5, each verb entry contains additional information about how a sentence should continue after the appearance of said verb. This information is used to expand the basic grammar pattern list after each verb in it. Verbs can be succeeded by objects, prepositions and objects, infinitive verbs, clauses and so on. Depending on the choice of the verb, the list is updated with the necessary POS-tags, in the correct location. In case of a preposition, the verb entry in the lexicon also contains a information about the possible prepositions that could appear with the specific verb and thus a certain preposition is added to the grammar pattern list. Any additional modifications required for the added word types are also managed automatically, as will be discussed in detail in 6.

4.1.2 Grammar Correction

After ensuring that the sentence structure itself is correct, some further grammar rules need to be enforced. This additional corrections are preformed outside the grammar pattern list, as they depend on other word choices. For instance, the article can only be correctly set after the noun itself has been chosen. In case of a plural noun, the article is changed to 'the'. In case of a noun starting with a vowel the article becomes 'an'. And finally, if the chosen noun is a pronoun, then the article is entirely removed. Another case were the grammar correction needs to be handled explicitly, is the s-form verbs.

Whenever the noun of the sentence is a singular third person the verb types in the whole grammar pattern list have to be modified to become s-form verbs instead of regular ones. In case of a past verb, this restriction is naturally ignored.

4.2 Poeticness

In this section the realization of the poeticness feature will be discussed; i.e. how we ensure that the generated texts are poems. The interesting thing about poetic features, is that they are many and that now all of them have to appear in a certain poem at once. Also, they are not very strictly defined and restricted like other linguistic features and are highly dependent on subjective taste and opinion.

4.2.1 Basic Features

One feature that most linguists agree upon, for a text to be considered a poem, is the rhythm. The rhythm of the poem is realized similarly to the grammar of the poem: through a metre list. However, instead of defining the target metre list of the whole poem at once, the list is defined for each verse separately. This is so, because depending on the poem type, each verse can have a different rhythm, form and length. The target metre list, is defined based in the required rhythm pattern of the poem. It consists of zeros and ones to represent stressed and unstressed syllables. This enables, the pattern matching with the rhythm pattern lists of each word in the lexicon. The pruning of the candidate words according to the rhythm, is performed by the reasoner after the selection of the POS-tag matching words, as shown in **??**. The finishing of a single rhythm pattern list, signals the termination of a poem's verse, which denotes the final chosen word as the last word of said verse. This brings us the the other popular feature of poems: rhyme.

If a user decided he wants to enable the rhyme in the generated poetry, an extra constraint is enforced on the last word of each verse. Depending on the chosen rhyme scheme the last words of certain verses have to rhyme. In these cases, the reasoner narrows down the list of possible words further to allow only suitably rhyming last words. To decide whether two words rhyme, the phonemes following their last stressed vowels are checked. If they match and the consonants preceding the vowel are different, then the words are considered to rhyme. Rhythm and rhyme alone, are enough to transform a text into a poem.

4.2.2 Figures of Speech

But the actual beauty of poetry comes from the added figures of speech that make for for the individual interpretation and imagination of each human. While it is possible to investigate the deliberate incorporation of figures of speech, in particular metaphors,

the chosen approach focused on examining, how metaphors could appear through underspecification. So instead of following the human's definition of what a metaphor is and how it should be, the computer was allowed to generate his own metaphors. This was achieved because the words were somehow chosen randomly, as long as they satisfied the harder constraints of grammar, rhythm and also semantics. The metaphors that appear in the generated poems are different from the accustomed human generated metaphors, which allows for more creativity and interpretation. While metaphors are defined as linking two different planes with each other Hobbs (1979), the system allows for metaphors that span higher dimensional planes and unconventional connections that humans would usually shy away from. An example of two metaphors generated in poem snippet is: `'Safety tears a tear,or feels for a fear'`. The reasoner is choosing rhyming and rhythm matching words, that have no other restriction than the grammatical structure. Yet, it produces two metaphors that can be interpreted by the reader, because of the random word choice governed by coherence (which will be discussed later). Although, metaphors are generated as a by-product of the generation design, it is possible to manipulate them upon requirement. This can be done by enforcing the linking of two different theme-based lexica at certain location. This is usually enforced with appearances of the comparative 'like', where the word preceding it should come from the indigenous lexicon, and the succeeding word should come from the target lexicon. The target can either be specified by the user, chosen randomly or chosen based on commonness of use with the indigenous lexicon.

Enjambments are also implicitly achieved in the generated poems, because the grammar patter list is globally defined for the whole poem, while the rhythm pattern list is only defined locally for each verse. So again, through underspecification, enjambments are generated because a sentence could span two verses.

Some other phonemic figures of speech that can be explicitly specified and incorporated into the system are repetitions, alliteration, anaphora and assonance. Repetitions can simply be enforced, by specifying a certain word, sentence or structure, that should be copied again at certain locations of the poem, to appear in certain intervals. A similar concept applies to anaphoras. Alliteration can be achieved by applying an extra constraint on the word choice, for the whole poem or for parts of it. The extra constraint would require two words to start with the same letter and exclude all the other words. Similarly, assonance can be achieved, but by putting a constraint on the vowels of the words instead.

4.2.3 Form

The final important trait of poems, is that they must have a unique form. Poems can have various different forms, which can be specified by the rhythm pattern list. One clear example of poetry forms, is concrete or shape poems. For example, if the user requires the generation of a diamante, the rhythm pattern lists would have the following form:

```
[_,_],[_,_,_,_],[_,_,_,_,_,_,_],[_,_,_,_,_,_,_,_,_],[_,_,_,_,_,_,_],[_,_,_,_],[_,_]
```

Similarly, the poem's form can be manipulated in many ways.

Also, some poems have certain features or structures, where they describe a certain environment, action or concept. This can be manipulated by the grammar pattern list and the rhyme pattern list together. For example, for a descriptive poem, it could be enforced that it starts with the thing it wants to describe and that after that no new subjects appear throughout the whole poem and only verb sentences that describe the main subject of the poem appear. Another possibility is to enforce the poem to start with specific actors or things and then move on to a more general case, or vice versa. Generality can be achieved by concepts and nouns describing broad things, while specification can be realized through persons and items. The type distinctions between the nouns, is enabled because of the specific lexicon design as will be shown in 5

4.3 Meaningfulness

The final and most ambiguous feature of poetry, and any literature text in general, is the meaningfulness. Meaningfullness is difficult in that it cannot be specifically defined. For the purposes of this work, it has been enforced through the concept of coherence, because a text can only be meaningful if it is coherent. Coherence has been ensured because of two measures, the generation of theme-based lexica and the restriction of the actors in the poem.

4.3.1 Theme-based Lexica

The main notion of coherence is achieved because of the extraction of smaller theme-based sub-lexica from the general lexicon. By enforcing a poem to be generated solely (with the exception of intended metaphors) from a lexicon that contains words belonging to a specific theme, it is ensured that the poem will have this theme. Also, all the words in the poem will have a certain unity in terms of the topic and the text will gain a meaning. This also allows for freedom in the meaning of the generated poems, so that they do not sound too constrained but allow the reader to interpret the actual message of the text as he pleases.

4.3.2 Poem Actors' Restriction

Another measure taken to improve the coherence of the produced poems, is the restriction of the actors of the poem. In other words, to limit the number of subject and objects that act in one poem. This is achieved by starting out with one subject. Then, whenever choosing another subject or object, there are two choices: either choose from the list of existing subjects and objects or choose a new one and add it to the list. The decision between the two options is done randomly with higher probability for the existing subjects

and objects. This allows the poem to have more coherence and unit, as it specifies the individuals the poem is revolving around.

Some extra restrictions can be enforced at this point. There is the soft constraint of the uniqueness subject and object of the a certain sentence. This constraint is thus not always enforced, to avoid excluding sentences that would provide a poetic meaning, where the subject of the sentence acts upon himself. Also, the exact number of subject and objects of the poem can be specified. For example, a certain poem can have one specific subject and one object. Additionally, the user could specify the actual actors (subjects and objects) of the poem. So, one could generate a poem whose only actors are a 'king' and a 'queen', for example.

Chapter 5

Lexicon

5.1 Design

The poetry generation process usually needs a lexicon or a corpus to provide the system with the words. In the introduced system the reasoner consists of pruning rules, that are applied on the lexicon, to generate each word. The lexicon is designed specifically to simplify the word selection and pruning process, by providing the reasoner with the needed information, in the format best suited for the implementation using CHR. The lexicon is generated using Java and CHR, by merging and reformatting University (2014) and Beale (2014), before adding some external information. Alternately Wordnet University (2010) could be used, instead of using Beale (2014), al Each entry in the lexicon has the following format:

```
word type;number of syllables;stress pattern list;pronunciation list;word
```

Where the word type represents the POS-tag of the current word, the number of syllables corresponds to the number of vowels in the word and the stress pattern corresponds to the number representing the stress of each vowel phoneme in University (2014). The pronunciation list contains the phonemes for the pronunciation of the word, based on University (2014). The final element in the entry contains the actual word. For example, the initial entry of the word 'charm' would be `verb;1;1;ch,aa1,r,m;charm;`. Initially, we differentiate between 7 different word types: noun, verb, adjective, adverb, pronoun, preposition and conjunction. Having the type of the word at the beginning of each lexicon entry, enables us to only import those words from the lexicon that belong to the required POS-tag. The first pruning step is thus preformed in the lexicon file, with the instructions of the reasoner. This makes the CHR pruning rules more efficient by having a much smaller candidate list to operate on, as will be shown later.

The described information contained in the lexicon, is already enough to allow the reasoner to realize the poeticness features: namely rhythm and rhyme, because of the

available stress pattern and pronunciation list. The basic notion of grammaticality is also achieved, in that the chosen words will always match the word type at the head of the grammar pattern list, regardless of how the latter is generated. A sample output generated by using this lexicon is:

the maid amends except the numbers
flogging till linguist solely summons
contrite charades detailed or raped
import diseased or tipster shaped

Although the text sounds like a poetic construct it definitely does not make much sense. One can easily notice the weird sentences and lack of coherence in the text. Thus, the first modification made to the lexicon is narrowing it down to get rid of the words that are not very common as well as those not likely to appear in poems (e.g. 'tipster'). Also instead of always considering the whole main lexicon whenever generating a poem, smaller theme-based sub-lexica are generated and only one of them is considered for the creation of one poem. This improves the coherence of the poem because only words related to each other, with respect to the theme, will appear in the candidate list produced by the sub-lexica. For the time being, this is done manually, to prove the concept of coherence on the sample theme of love. Using the love-themed lexicon instead of the main one, would generate such poems:

friendship breathed or brightly trusted
lovely marriage wedded and divorced
a wife touching after dearest lust
firing over music and caress
tear adored and kindly missed
lone affairs divorced and kindled
a boy dreaming on devout romance
never dreaming men behind eclipse

While the poem sounds like a construct that could be produced by a human, we notice, for example, the odd sound of the use of the preposition 'after' with the verb 'touching' or 'on' with the verb 'dream'.

This is why the lexicon is further enhanced, to add the necessary information needed to construct the rest of the sentence, after the appearance of a verb. The single entry of the verb 'charm' is now extended to `verb;1;1;ch,aa1,r ,m;charm;obj` and `verb;1;1;ch,aa1,r, m;charm;obj,prep, ing;into`, each time conveying a different meaning. The first occurrence of the verb 'charm' states that the sentence can be continued by adding an object after the verb. The second gives the possibility of succeeding the object with a preposition and a verb in its 'ing' form. However, we need to define exactly which prepositions can be used, which was the reason for this extension in the first place. Thus, another element is added to the entry, namely the list of possible prepositions; in

the case of the verb ‘charm’, ‘into’. The reasoner regards the two entries as two different ones of the same POS-tag.

In order to be able to generate poems using the extended lexicon, some modifications need to be applied. Further type distinctions are incorporated into the lexicon. We differentiate between different verb forms e.g. the ‘-s’ and ‘-ing’ form and different verb tenses, namely past and present. In the case of nouns, we differentiate between: actor, concept, thing, item and plural. While the subject of the sentence is initially restricted to be of type ‘actor’, the object of the sentence can be of any noun type. Using the new extended lexicon, it is possible to generate such sentences correctly:

A Miss felt alive about the hug
She sadly charms the boy into singing

5.2 Generation

The first version of the lexicon, which is a merging of various linguistic resources, was generated using Java. Additional enhancements to the lexicon were realized using CHR. The remaining needed information was preliminarily added manually to a small sample lexicon, as proof of concept. In the future, this part is to be automated and generalized to cover the whole lexicon.

5.2.1 Java

The initial main lexicon is a compilation of two main resource files, namely the ‘cmudict.txt’ file in University (2014) and the ‘2of12id.txt’ file in Beale (2014). Before starting to merge the files together, some minor changes were performed on them, in order to make their future use easier. The double spaces separating a word from its pronunciation, were replaced by a single space in ‘cmudict.txt’. Also, to reduce diversity, unnecessary inflections and differentiations of words were removed from ‘2of12id.txt’. All words preceded by a single hyphen or a ‘∼’ were also removed, due to being unusual. Furthermore, words of type ‘spoken’ or ‘interjection’ are ignored, as they are not very familiar in poetry and would require special treatment.

At this point, both files, ‘cmudict.txt’ and ‘2of12id.txt’ are ready to be read by Java and the needed information is extracted from them to be written into the output lexicon. As discussed, each entry in the produced lexicon, has the following form: `word type;number of syllables;stress pattern list;pronunciation list;word`. In the following we will always consider each lexicon entry, as a list, whose elements are separated by a ‘;’ and we will refer to the elements by either their name or their number in the list. The semicolons separating each element of the lexicon entry allow the Prolog predicate `transform/2` to transform each element in candidate list into a 2D list of the form `[[type],[no of syllables],[stress pattern], [pronunciation]]` to be able

to access each element of the entry using its corresponding index. The advantage of this notation will become clearer when we go into the details of the poetry generation process. The actual word is appended at the end of each entry, to be displayed faster in the final poem, which will also be shown later. The 'my2of12id.txt' is read to provide the word categorizations and the 'cmudict.txt' file is read to provide the pronunciation, syllable number and stress list.

A `Word` object is created for each line from the 'cmudict.txt' and is stored in a list of `Words`. Each `Word` has two attributes, the word itself and the pronunciation list. For each line of the 'my2of12id.txt' a `WordCategory` object containing the word and its category is created and stored in a similar list of `WordCategories`. For each instance of the class `WordCategory` we gather all the necessary information and create the output line for the lexicon, in the desired format.

The `process` method in the class `WordCategories` is responsible for this procedure. Depending on the category of each word in the `WordCategories` list, it adds the word type to the string to be written into the output line. It then accesses the word with the same index in the Words list, to extract the remaining information. To do so it makes use of two helper methods, namely `syllableCounter` which is responsible for counting the syllables of the current word by counting the number of phonemes starting with a vowel and `stressPattern` which is responsible for creating the stress list of each word by checking on the number appended to the vowel phonemes. It is important to note, that for the purposes of the poetry generator, we only need to distinguish between stressed and unstressed syllables, without needing the strength of the stress, thus both stress values 1 and 2 are stored as a 1 in the stress list. After having calculated the syllable count and the stress list they are also appended to the output string. The last information to be added to the string before the actual word in question is the pronunciation list. All the information concatenated to the string is separated by ';'. Whenever a string is ready, it is added to list of strings to be written back to the output lexicon.

Due to the fact that inflections of the same word are normally stored in the same line in the 'my2of12id.txt', and are thus all stored as one word in both the `Word` and the `WordCategory` lists, these cases need to be handled separately. The first case where this is relevant is in the case of adjective where each adjective usually has two superlatives. If they exist, the information for each of the superlatives is gathered and stored into individual output strings. However, we do not differentiate between the three types of adjectives and store all three as words of type adjective. The second case is that of nouns, where the plural of a noun is stored right after it in the line, if the noun has a plural form to start with. In this case the info for the plural noun is gathered at the same time and stored with the type plural in a line of its own. The final case for this occurrence is for all the verbs, where each verb usually has multiple inflections for various tenses. Each line in 'my2of12id.txt' contains these inflections in a certain order. So each line is scanned and the first word alongside its relevant information is stored with the type verb, the second one with the type past, the third as '-ing' form and the fourth as '-s' form.

After all strings are created and formatted correctly they are written into an output file that serves as the first draft of the lexicon. Initially, only words with a maximum of

two syllables are written into the lexicon, seeing as to how the generated text will be a poem with a limited verse size.

5.2.2 CHR

The produced lexicon is then passed on to be processed by the CHR code. The role of the CHR code is to differentiate between adjectives and adverbs with the help of the `s/6` predicate of the Prolog version of the Wordnet database University (2010).

To start, the UNIX `grep` command is called from within CHR using the predicate `grep`, to get all the lines of the lexicon file, generated by the Java code, and store them in a list in the constraint `all_words/1`. We then loop on the elements of the list. Whenever we find a word of type adjective, we check on the type attribute of the `s` fact corresponding to the current word. If it has the letter 'r' as its type, then the whole line should be replaced in the lexicon as the word should have the type adverb instead. After the necessary additions and removals have been made to the list and all lines have been checked, meaning the base case has been reached, the whole list is written into a new lexicon file.

5.2.3 Additional Information

Before adding the additional necessary information, a smaller love-themed lexicon was generated, by selecting all word related to love from the main lexicon. As discussed before this process, should be automated in the future, as it was applied to a small sample as prove of concept before being generalized. The additional information added to the sub-lexicon will be discussed for each word category on its own.

5.2.3.1 Nouns

First of all further type distinctions between different nouns are made. Nouns are divided into actor, concept, item, thing and plural. This would enable the enhancement of the meaningfulness of the poem, whenever needed, by restricting the interacting noun types of a sentence or a stanza.

5.2.3.2 Verbs

Depending on the verb type, each verb can be succeeded by a different sentence structure. For each verb we then add different entries for each possible continuation of the sentence, after said verb. If the sentence continuation list contains a word of the type 'proposition', then a list of the possible propositions, that can be used, is also added. A verb entry line thus has the form:

```
verb tense;number of syllables;stress pattern list;syllables list;verb;
sentence structure list; prepositions list;
```

Each possibility of a continuing the sentence after the verb is stored in a line of its own as each one will be handled as a different possibility whenever the poetry generator is choosing its words. As each verb can be continued with a different sentence structure, as discussed, each verb line contains different information; e.g. not all verbs will have a propositions list. Due to this fact, we need to add semicolons representing the sixth and seventh elements of any verb entry, even if these elements are empty and not needed for the current verb. This is done, in order to avoid the unnecessary failure of reasoner queries involving verbs when generating a poem.

5.2.3.3 Pronouns

Pronouns have types of the form `'pronoun$\_$letter$\_$person'`, where the `letter` is either 'o' for object pronouns,'a' for possessive adjective, 'p' for possessive pronouns and 'r' for reflexive pronouns; representing the four pronoun types. The `person` entry consists of the person number, followed by either the letter 's' or 'p', for singular and plural persons, respectively.

The remaining pronoun type, namely the subject pronouns, are stored with a different type format than the remaining pronouns; 'actor_pronoun'. This is so, because these pronouns can be used as actors and thus need to have the prefix 'actor' to their type, in order to be included in the search for actors by the reasoner. Information about the acting person is stored in the 6th position of each of these subject pronouns, as it is important to be able to differentiate between singular and plural actors, as well as actors in the third singular person, which require the verbs following them to be in s-form.

Also, all pronouns have an additional element at the very end containing either 'm', 'f' or 'n' for masculine, feminine and neutral, respectively.

5.2.3.4 Prepositions

In case of prepositions, the word 'preposition' is appended after the preposition itself-separated by an underscore- in the word type entry. The type of each preposition is prefixed with the actual preposition, as prepositions cannot be interchanged and an exact match needs to be achieved whenever searching for a preposition to use.

Also some prepositions are subsets of each other, such as 'in' and 'inside', causing the reasoner to receive both 'in' and 'inside', when requesting 'in', as the 'grep' predicate only looks for lines starting with a certain substring. To avoid this mismatch, prepositions that are subsets of others are prefixed with the letter 's' in their type. The type of the preposition 'in' would thus be 'sin_proposition'. It is important to update the preposition lists of the verbs, to compensate for the changes in the preposition type. A verb

requiring the preposition ‘in’ would instead have ‘sin’ written in its preposition list; e.g. `verb;1;1;t,r,ah1,s,t;trust;prep,obj;sin;`.

Some verbs require two prepositions after them but only have one preposition list. For these cases individual entries are created where the preposition type has both prepositions as a prefix but the word element itself has them both separated by a space so that they appear correctly whenever they are displayed in a poem. An example of such an entry is `upto_preposition;2;1,0;ah1,p,t,ah0;up to;`.

5.2.3.5 Articles

The only article added to the lexicon is ‘a’ because the other differentiations, namely ‘an’, ‘the’ and possessive pronouns are handled by the poetry generator itself.

Chapter 6

Implementation of the Reasoner

In this chapter we will discuss the realization of the solution approach discussed in the previous chapter, through CHR.

Constraint	Meaning
`candidate(V,I,W)`	`W` is the final candidate for position `I` of verse `V`
`word(V,I,W)`	`W` is the actual word in position `I` of verse `V`
`last_word(V,I)`	The word with index `I` is the last word in verse `V`
`pattern(V,G,S)`	`G` and `S` are the remaining grammar and stress pattern lists of verse `V`, respectively
`count(V,I)`	The word with index `I` is the current word to be processed in Verse `V`
`suObject(S)`	The list `S` contains the subjects and objects of the poem

Table 6.1: List of Constraints

After having described the lexicon structure and its generation, we will go into the details of the reasoner component. The main constraints used by the reasoner are described in Table 6. The CHR rules define the interaction of the main constraints with each other and the application of the actions (prolog predicates) on them, in order to prune the candidate list; all the while satisfying all the poetry constraints.

6.1 Initialization and Termination

Before starting to generate any poems, the system has to add some initial constraints, which are considered the internal inputs of the reasoner. First, the grammar pattern list `G` is added to the constraint store, to be used as a basis for the poem generation. The list can be given the exact pattern required in advance; however, that would be

very tedious and would always generate similar poems. However, the format of the verb entries in the lexicon allows us to know how the sentence pattern should continue after any chosen verb. Thus, only subject and verb are given to the grammar pattern list. Also whenever a conjunction needs to be added between two sentences this should be stated specifically in the grammar pattern list. An article is forced to always appear before a noun, but it can be retrospectively removed or modified, depending of the final noun chosen. In order for the poetry generation process to start, we need to generate the constraint `pattern(V,G,S)`. A sample grammar pattern list `G`, representing a poem could be: `[article,actor,verb,conjunction,article,actor,verb]`. The stress pattern list `S`, contains the stress pattern of the number of syllables, allowed in verse `V`. The `count/2` constraint is also needed, as it keeps track of the number of words allowed in each verse. Note, that the stress pattern list is only equivalent to the current line, while the grammar pattern list spans the whole poem. Thus, whenever the stress pattern list is empty (more details in the following), we initialize a new `pattern/3` constraint for the following verse, with the remaining elements of the grammar pattern list and a new stress pattern list. The CHR rules, enable this separation between the two lists and provide for more flexibility in realizing the required features of poems. For example, if we choose that the poem will follow a monotone rhythm, the same initial stress pattern list is given to the `pattern/3` constraint, of all the verses. Whenever a new verse is started, a `last_word/2` constraint, with the previous verse number and the count index it had reached, is added. The poem is only fully generated, whenever the grammar pattern list becomes empty. As the current implementation of CHR follows the refined operational semantics, the rules will always be applied from top to bottom. Thus the final rule is the one responsible for generating a `word/3` constraint for any candidate word that has passed through all the rules. Although the `word/3` constraint symbolizes a final poem word, it can still be changed retrospectively whenever needed, which shows the flexibility of using CHR.

6.2 Basic Grammaticality and Poeticness

The actual poetry generation process, where all the pruning occurs, consists of two basic rules:

```
Rule 1 @ pattern(Verse,[H|T],TargetMetre), count(Verse,Count)
<=> grep(lexicon, H, TypeCandidates),
narrow_down(TypeCandidates,TargetMetre,MetreCandidates),
random_choose(MetreCandidates,Candidate),
candidate(Verse,Count,Candidate), update.
```

Rule `1` consists of two pruning steps. The first one occurs, while selecting the words from the lexicon, using the `grep` predicate; by only "grepping" lines that start with the current head of the grammar pattern list. The resulting list contains the so called TypeCandidates. The second pruning step is realized, by the `narrow_down` predicate,

which takes as input the TypeCandidates and the TargetMetre(the current stress pattern list), and produces the MetreCandidates list. The MetreCandidates list is generated by only keeping words from the TypeCandidates, whose stress pattern is a prefix of the TargetMetre. Now that all the remaining words are viable, we randomly choose one of the words to become a candidate for the current position. The `random_choose` predicate allows for backtracking, to try out a different words, in case no solution can be reached. The update constraint is a black-box for the remaining maintenance operations that occur in the rule. That is, the update of the stress pattern list by removing the number of syllables used by the chosen word and the removal of the used head of the grammar pattern list. Finally, the whole `pattern` constraint is updated with the new lists.

```
Rule 2 @ candidate(Verse2,ID2,ExWord2)\ last_word(Verse2,ID2),
candidate(Verse1,ID1,ExtWord1), last_word(Verse1,ID1)
<=> Verse1 is Verse2+1, Verse1 mod 2 =:= 0 |
get_type(ExWord1,Type), grep(lexicon,Type,TypeCandidates),
narrow_down_rhyme(TypeCandidates,ExWord2,RhymeCandidates),
random_choose(RhymeCandidates,Candidate),
candidate(Verse1,ID1,Candidate), update.
```

Rule 2 handles the verse ends, where we need to enforce the additional constraint of rhyme, when choosing a word. Here, we notice how important the multi-headed rule matching really is, as we only want this rule to be applied, whenever two candidate words, that are also in positions indicated by the `last_word` constraints, exist. Also the guard conditions, ensuring that the two words, are in two verses that should actually rhyme, should also be satisfied. For example the shown guard conditions represent the rhyme scheme: 'aabb..'. Because we now need to replace a word, with another one of the same type, we need to access the type of the candidate word to be replaced. After the two pruning steps explained in the rule 1 have been applied, we preform the final pruning step; namely extracting the RhymeCandidates from the TypeCandidates. The `narrow_down_rhyme` predicate checks, if the words, in the TypeCandidates, rhyme with the other existing word, and adds those to the RhymeCandidates. This is done with the help of the `rhyme` predicate, which takes the pronunciation list of two words and locates the final stressed syllable (i.e. with stress 1) and the consonant preceding it. Two words rhyme, if all their syllables, starting the final stressed syllable, are identical, and the consonants, preceding said final stressed syllable, are different.

These two rules alongside the lexicon are enough to produce poems, as discussed in Chapter 5. The poem presented before is generated using only those two rules:

the maid amends except the numbers
flogging till linguist solely summons
contrite charades detailed or raped
import diseased or tipster shaped

6.3 Additional Grammatical Constraints

The basic rules defined above, coupled with a complete predefined grammar pattern list, are sufficient to generate poetic texts, as shown above. However, we had mentioned, that only a basic grammar pattern list will be input and the expansion will occur automatically. This extension reduces the number of external data needed for the reasoner, making it more autonomous. The grammatical quality of the generated poems is also improved.

```
Fix Rule @ count(Verse,Count),candidate(Verse,ID,ExistentWord)\
pattern(Verse, [Type|T],TargetMetre)  <=> (ID is Count+1;
ID is Count+2), get_type(ExistentWord,ExistentType),
fix(Type,ExistentType,FixedType),
pattern(Verse, [FixedType|T],TargetMetre).
```

This rule applies most of the modifications to the grammar pattern list in advance, whenever a new candidate is chosen, in order to ensure that any chosen word is correct to start with. This makes the generation process more efficient. This is also necessary, because the selection of each word depends on the existing candidate words and thus, any change in the already chosen words needs to be carefully conducted as not to disrupt the rhyme and rhythm, in particular. One example modification enabled by the above rule, is replacing the type of a base form verb with an 's' form one, whenever the subject of the sentence is in the third singular form. This is done by checking the the type of the existent word, that, either directly or indirectly, precedes the word to be chosen. This is to take into consideration, the possibility of the existence of adjectives between a subject and a verb. The `fix` predicate, fixes the type of the words to be chosen, according to the type of the existing word, if necessary. The modified word type is added to the stress pattern list, and the generation process can proceed normally. Another important modification is the expansion of the grammar pattern list following a verb. Whenever a candidate word of type verb is selected, some rules similar to the `fix` rule, apply additional modifications to the grammar pattern list. Such verb-specific modifications are:

- Adding the grammar continuation list of the verb to the existing grammar pattern list
- Replacing words of type preposition with the possible prepositions depending on the verb
- Randomly replacing an object with one of the different noun types
- Randomly deciding to place an adjective or and adverb before any noun
- Extra restrictions on sentence structure

Other than the case of backtracking to find a rhyming word pair, there is only one instance where the retrospective fixing of already placed candidates occurs. This happens when dealing with articles because an article precedes a certain word and thus can only be modified after said word has been chosen. In case of pronouns the article is removed, while words starting with a vowel require the replacement of 'a' with 'an', and plural nouns sound better when preceded with the article 'the' instead of 'a'. These fixes are relatively simple because the constraints can be replaced using the CHR rules, and any additional updates are automatically handled by the reasoner.

These additional fixes can produce poems and English sentences of of higher quality, like the ones presented in 5:

A Miss felt alive about the hug
She sadly charms the boy into singing

6.4 Additional Poeticness Constraints

In addition to the initial poeticness constraints, there are many extensions that can be implemented to enhance the poeticness of the generated poems.

1. Repetitions: The choice of some words could be skipped, and other already existing words could be put is these positions instead. This is manipulated by specifying the certain words that should be repeated, and then whenever choosing a word with the same type, the existing one is inserted. Another possibility is to choose to duplicate whole verses that are considered fundamental for the poem. The repetition interval can be specified. For example the verse can be repeated at the end of each stanza or after each verse etc.

2. Enjambment: Enjambments are a by-product of the chosen design of the reasoner inputs. As the grammar patter list spans the whole poem and the rhyme pattern is specified for each verse separately, some sentences can span multiple verses leading to enjambments.

3. Metaphors: As previously discussed, metaphors are actually automatically incorporated because of the nature of the implementation approach. However, they can also be explicitly enforced, if required. For the time being, this is done for the subset of metaphors with the comparative 'like'. Whenever the word 'like' is chosen by the reasoner it is ensured that the word following it is from another theme and that a direct metaphor is generated. It can be specified that the two words linked by the comparative are of the same word type or this constraint can be relaxed to allow for more creative metaphors.

4. Form: The form of the poem can mean two separate things:

(a) The actual form of the poem, which consists of the number of verses and stanzas, their length, the rhythm and rhyme scheme etc. All these can be specifically defined and manipulated by the grammar and stress pattern lists and the chosen rhyme scheme

(b) The narrative form of the poem, meaning the flow of the story expressed by the poem. Poems usually have a specific thematic construction or setup. For example a poem can start by talking about a general situation and then apply it to specific individuals, or vice versa. Also a poem, could be a detailed description of an environment or a specific thing. It could be a story, in case of a ballad, and so on. Some of these structures can be implemented by the reasoner, through enforcing extra constraints on the structure of the grammar pattern list, as this deals with the choice and order of the word types in the poem. As mentioned before, this is one major use of structure of the lexicon.

6.5 Coherence

An extra step that can be taken, in order to improve the coherence of the poem, other than the theme-based lexica, is putting a constraint on the choice of the subjects and objects of the poem. This is achieved by the following rule:

```
Coherence Rule @ suObjects(A), pattern(Verse, [H|T],TargetMetre),
count(Verse,Count) <=> H = 'subject'; H = 'object' |
choose_candidate(C), append(C,A,A1),
random_choose_unique(A1,Candidate),
candidate(Verse,Count,Candidate),
(suObjects(A);suObjects(A1)), update.
```

Each time a new subject or object is to be chosen, a candidate is chosen normally, following `Rule 1` and `Rule 2`. Then, it is added to the list `A`, which contains the subjects and objects of the poem, so far. From this list, one element is chosen randomly and it is used as the subject or object of the list, respectively. The only restriction that can be applied on the choice, is checking that the subject of the sentence is not the same as the object, which is done by the predicate `random_choose_unique`. However, this constraint can be intentionally relaxed, or if no solution is found with this additional constraint. If the candidate `C` was used as the subject or object, then the new list containing `C` is added, else the old one is kept. This procedure, ensures that the number of subjects and objects acting in the poem is limited, without sounding too constrained; which adds more coherence to the flow of the poem. The number and type of the subjects and objects can also be specified and set. Additionally, the exact subjects and objects can be chosen beforehand through the `suObjects` constraint. The user can either decide to add additional actors to the specified ones or restrict them to the actors he chose. The restriction of the number of acting subjects and objects of the poem, could cause some

minor discrepancies in the rhyme of the poem, but they are negligible, in comparison to the achieved improvement in the poems' quality. The following poem, that was displayed in 5 shows said improvement:

friendship breathed or brightly trusted
lovely marriage wedded and divorced
a wife touching after dearest lust
firing over music and caress
tear adored and kindly missed
lone affairs divorced and kindled
a boy dreaming on devout romance
never dreaming men behind eclipse

6.6 Choices

The user can manipulate everything about the poem that should be generated. The high freedom in the manipulation of the poem's features is enabled because of the nature of CHR. The different requests, simply mean the application of certain constraints through their respective rules and the relaxation of some others. Without performing any changes to the generation code, minor specifications, are enough manipulate the generated poem. As mentioned throughout this chapter and previous one, the grammar and rhyme pattern lists alone, play a powerful role in generating different types of poetry and in expressing the various poetic features. Depending on the required poem type and characteristics, the system chooses from a set of predefined grammar pattern and stress pattern list pairs, to generate the required poems. For example, if a sonnet should be generated the system initializes a rhythm pattern list with the exact number of verses of the Shakespearean or any other sonnet type and sets the rhythm and rhyme scheme to corresponding ones. It then generates a grammar pattern list that would contain enough sentences to fill the whole sonnet. It should be noted, that because of the separated nature of the grammar and rhythm pattern lists, it could sometimes be the case that the required poem form is not exactly reached, if the grammar pattern list is consumed prematurely, before all the required rhythm pattern lists have been used. The opposite could also happen, where rhyme lists become empty before the whole grammar pattern list has been used, which could lead to sometimes uncompleted sentences. However, this could be handled by adding fail-safe constraints to avoid these situations or fix them if they arise. Also, additional specific grammar and rhythm pairs could be specified by the user and the reasoner would generate the poems accordingly. The user can also choose the theme, actors and specific words that should appear in the poem. The user has all the freedom to add or define specific metaphors or metaphor structures, he wants to appear in the poem.

Chapter 7

Semantics

7.1 Concept

The quality of the generated poems using CHR so far, has been proven to be comparable to human generated poetry. So far, the meaningfulness feature has been realized through coherence alone and the rest of the meaningfulness was realized through underspecification. Our interpretation of meaningfulness can however be extended to semantic relatedness. The reasoner was enhanced by adding a semantic function that chooses the semantically related words, that match the other poetry constraints instead of the performing the final word selection randomly, as shown in 4. To achieve the highest efficiency, the semantic network of each word is generated beforehand and stored externally in a file, with the word name. This highly optimizes the generation process, because the semantic relations list can be generated by reading the information from the specific pre-generated files, which saves the repetitive execution of the same task.

7.2 Semantic Network Generation

In this section we will discuss the generation of the semantic network files of each word. To prove the concept, this was initially only performed for nouns of type 'concept', before extending it to all the words that appear in each theme-based lexicon. It should be noted, that the generation of semantic networks, would only be meaningful for certain word types. For example, pronouns, conjunctions and similar words do not need a semantic relations network. the generation of the semantic networks is also implemented in CHR, to extract the necessary information from the Prolog Wordnet files, which makes the task very straightforward. The algorithm takes a specific base word denoted by the `base_word/1` constraint and starts adding related words for it, through the `related_word/1` constraint. After all the related words have been found, they are added to a relations list which is written to a text file, that has the name of the base word. In Wordnet, the words are identified by their IDs or sense keys, as each word can belong

to different synsets and thus has multiple IDs. The first step after adding the base word, is getting the all its IDs from the different synsets and adding `related/1` constraints for them. The `related/1` constraint is the main constraint used for the generation of the semantic networks. At this point, all the information required for finding the words related to the base word, has been acquired , and the actual generation can begin. Due to the format of the Prolog files of Wordnet and the choice of CHR as the programming language, the related words, are found only by accessing the relevant Wordnet files, containing the facts of predicates representing the different relations given by Wordnet. The general rule used for this procedure is:

```
Semantic Relations @ related(ID), operator(Operator) ==>
wordnet_operator(Operator,ID,Rel_ID) | s_short(Rel_ID,Rel_W),
related_word(Rel_W), related(Rel_ID).
```

For every `related(ID)` constraint, if the Wordnet operator `Operator` is defined for the specific ID, then the ID `Rel_ID` of the related word is returned, and the word `Rel_W` corresponding to `rel_ID` is extracted from the `s_short/2` predicate. This predicate is a shortcut version of the Wordnet `s/5` predicate, that only gives the words and their corresponding IDs with disregard to all the additional information. A `related_word` constraint is added for `Rel_W` and a `related` constraint is added for the `Rel_ID`. This allows the system to look for indirect relations. For example, if the operator in question is the antonym, then the antonym of the base word, should be considered as a new word for which the related words should be found. However, this would lead to a huge list of relations that trace every word to all its possible origins which would breach the concept of semantic relationship, for the purposes of our work. Because, we need the words to be closely related to each other if we want to improve the meaningfulness and the semantic coherence of the generated poetry. Thus, for some of the operators, only the `related_word` constraint is added while the `rel_ID` is discarded. It can be decided based on taste or different criteria, which operators are allowed to propagate their found words, and which should only be used for extracting direct relations. The `Semantic Relations` rule is a propagation one, because the `related(ID)` should be used to find the other relations of the word from the other Wordnet operators. The program can be run for all the required words, to generate their respective semantic networks. This also enables the modification and extensions of the semantic networks externally, without doing any changes to the reasoner.

7.3 Integration

In the following we will discuss the integration of the generated semantic networks, to enhance the semantics of the generate poems.

Whenever a `candidate` constraint of a word type with a semantic network is added the `Update Semantic List` rule is applied. Initially, this rule is only applied on words of type ‘concept’, but this can easily be extended to all relevant word types later.

```
Update Semantic List @ candidate(Verse,Count,Candidate)\
 semantic(L)<=> concept(Candidate)|
 grep(CandidateSemanticNetwork, RelatedWords),
 append(RelatedWords,L,L1), semantic(L1).
```

The rule is applied, if the `concept(Candidate)` hold, i.e. if the word is of type 'concept'. If this is the case, the whole semantic network `CandidateSemanticNetwork` in the file of the candidate is 'grepped' and stored in `RelatedWords`. This list is append to the already existing list `L` containing the semantically related words of all the words, that appeared in the poem so far.

So, whenever a final word should be chosen from a pruned candidate list, the random work selection is replaces by a matching with the list contained in the `semantic` constraint. The rules explained in 6 are all accordingly modified. For example, `Rule 1` will have the following form:

```
Updated Rule 1 @ semantic(RelatedWords) \
pattern(Verse,[H|T],TargetMetre), count(Verse,Count)
<=> grep(lexicon, H, TypeCandidates),
narrow_down(TypeCandidates,TargetMetre,MetreCandidates),
choose_related(MetreCandidates, RelatedWords, Candidate),
candidate(Verse,Count,Candidate), update.
```

The main modification, is thus in the predicate `choose_related/3`, which chooses a `Candidate` from the `MetreCandidates` that appears in the `RelatedWords` list, if it is found. If no match between the `MetreCandidates` and `RelatedWords` is found, a `Candidate` is chosen randomly, like in the original `Rule 1`. The smeantics is thus handled as a soft constraint to ensure the termination of the program. An optional measure for the semantic coherence could be given, where the value is increased each time a word has to randomly selected. The semantics of the generated poems, could be enhanced by locally considering the semantic relation for each sentence separately, instead of for the whole poem poem, at once. Also the semantic network itself can be extended in different ways to allow for more semantics. As mentioned before this can be used to improve the quality of the generated metaphors, by linking two different semantic networks to generate a metaphor. However, the current extension, is enough to prove the concept of improving the meaningfulness of the poem, through the incorporation of a basic notion of semantic networks.

Chapter 8

Evaluation

8.1 Quantitative Evaluation

As is any creative work of art, the evaluation of poetry is very subjective. It is hard to find a measure to define, what is good poetry, or even what is acceptable poetry. For the purposes of evaluating our work and achieving the aim of this thesis, a survey was conducted to quantitatively evaluate our work. The aim of the survey, is to prove that the poetry generated by our system is comparable to human generated poetry. Thus, we want to prove that the generated poetry is good enough to be generated by humans. The survey also aimed at trying to rate the quality of the generated poems, with respect to a set of features. The question whether the computer generated poems are better than human generated ones or not, is a purely subjective one, thus it will not be tackled here.

The survey consisted of 6 questions that cover different points, needed to support our claim. It was posted online on https://www.surveymonkey.com/s/8TNQMP2, and the results analyzed here have been collected over a one month period. A total of about a hundred opinions have been gathered. The partakers of the survey were unaware of the purpose of the survey or that automatic poetry generation was involved in order to avoid premature assumptions and biased opinions.

8.1.1 Survey

The first two questions of the survey presented 3 poems each, to be ranked by the user according to preferability. The first set consisted of one computer generated poem, and two poems by famous poets.:

1. *Tear adored and kindly missed,*
   Lone affairs divorced and kindled.
   A boy dreaming on devout romance,
   never dreaming men behind eclipse. [1]

[1] Computer generated

2. *I was a child and she was a child,*
   In this kingdom by the sea,
   But we loved with a love that was more than love
   I and my Annabel Lee
   With a love that the wingèd seraphs of Heaven
   Coveted her [2]

3. *Love is more thicker than forget,*
   more thinner than recall,
   more seldom than a wave is wet,
   more frequent than to fail. [3]

The second set consisted of one computer generated poem, and two poems posted online by amateur poets:

1. *It feels intense, so it misses,*
   to hide from a storm, still an adult kisses
   a hooked lip. [4]

2. *Kiss me like my lips are a forest fire.*
   And all your lungs need to breathe are flames [5]

3. *To lay a kiss on your lips,*
   so gentle and delicate
   is like picking the drops of dew
   off the petals of a rose. [6]

The second type of questions, is a comparison of two poems, to decide which one of them is generated by a human poet. Again the first set consists of one poem generated by the system developed in this thesis and the other is a poem by a famous poet.

1. *He trusts a Tear to sing along,*
   When he dances with the Hearts.
   He hates to trust in a Romance, where
   He loves to miss to dance with a Care [7]

2. *When Friendship or Love our sympathies move,*
   When Truth, in a glance, should appear,
   The lips may beguile with a dimple or smile,
   But the test of affection's a Tear [8]

[2] from "Annabel Lee" by Edgar Allan Poe
[3] from "Love is more thicker than forget" by E. E. Cummings
[4] Computer generated
[5] "Kiss" by WreckingballX, http://hellopoetry.com/words/210/kiss/poems/, 2014
[6] by Peter Oliveri, http://www.lovepoemsandquotes.com/LovePoem76.html, 2014
[7] Computer generated
[8] "The Tear" by Lord Byron

For this questions, the second set also compares the computer generated poem, with a poem or a poetic quote by a famous poet.

1. *Safety tears a tear,*
   or feels for a fear,
   heats a youthful love,
   or loves a hooked dove.
   Safety hides a dear
   breath, or heats a fear,
   hopes to break a tear,
   Or loves a fear. [9]

2. *Someone who does not run*
   toward the allure of love walks
   a road where nothing lives.
   But this dove here senses
   the love-hawk floating above,
   and waits, and will not be driven
   or scared to safety. [10]

After completing the first four questions, which aim at testing the comparability of the generated poetry with existing human written poetry, the reader is taken to a different set of questions. The second part of the survey, is designed to give insight about the quality of the generated poems. This is done in two parts:

The reader is given two computer generated poems and should choose from limited options, to give a rating for a certain set of features. The two poems are two from the ones used in the first question set, to keep the evaluation coherence:

1. *He trusts a Tear to sing along,*
   When he dances with the Hearts.
   He hates to trust in a Romance, where
   He loves to miss to dance with a Care

2. *Safety tears a tear,*
   or feels for a fear,
   heats a youthful love,
   or loves a hooked dove.
   Safety hides a dear
   breath, or heats a fear,
   hopes to break a tear,
   Or loves a fear.

[9] Computer generated
[10] by Rumi in "The Book of Love: Poems of Ecstasy and Longing"

The user rates the two poems in terms of:

- Rhythm (the tempo and flow): choice between 'yes' and 'no'
- Rhyme: choice between 'yes' and 'no'
- Message (the idea behind the poem) : choice between 'clear' and 'unclear'
- Language (is it understandable): choice between 'understandable', 'odd' and 'unclear'
- Style (the poetic style): choice between 'very poetic', 'somewhat poetic' and 'unpoetic'

The final question handles the figurative language separately. Here, the reader is presented only with the first poem of the two he or she just rated, and should decide whether it contains figures of speech? The reader can choose from the following options:

1. No figures of speech
2. Figures of speech of poor quality
3. Figures of speech of average quality
4. Figures of speech of good quality
5. Figures of speech of great quality

After answering the six questions, the user has completed the survey and the results are ready to be analyzed.

8.1.2 Results and Analysis

For the time being, only about a hundred opinions have been gathered and thus only a preliminary evaluation can done. In the following we will present the obtained results and analyze their implication on the work done in this thesis. The results of each questions will be analyzed separately:

1. Figure 8.1 shows that the computer generated poem was ranked first against the two other poems written by famous poets, by around 20% of the readers. In total around 60 % of the readers, ranked the computer generated poem, first and second, meaning considered it better than atleast one of the two poems written by famous poets. This shows that the poem is comparable to renowned poems and can sometimes even be considered of higher quality. This is the case, because the user is not suspecting that one poem should be scrutinized more than the others and is more receptive to imagination and creativity.

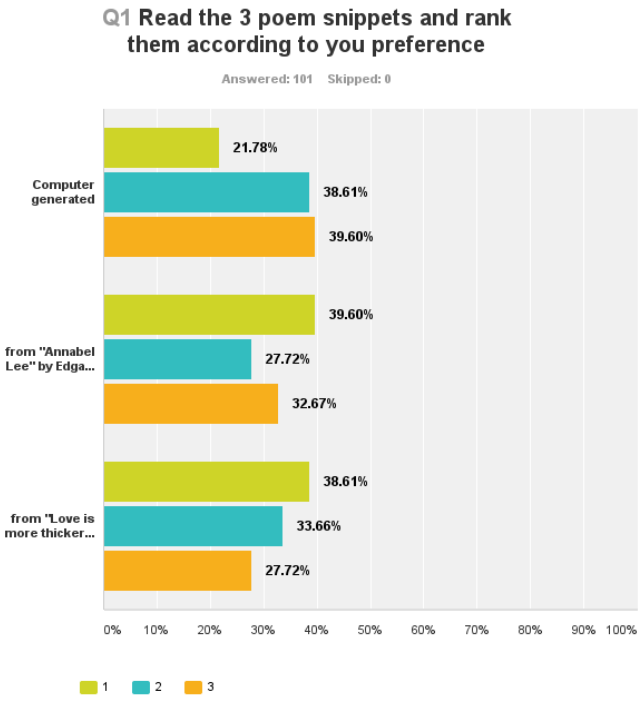

Figure 8.1: Results of Question 1

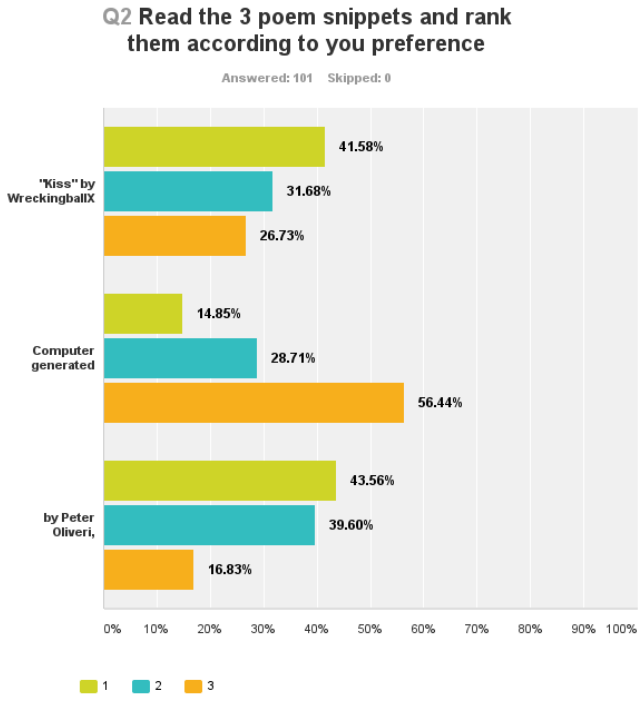

Figure 8.2: Results of Question 2

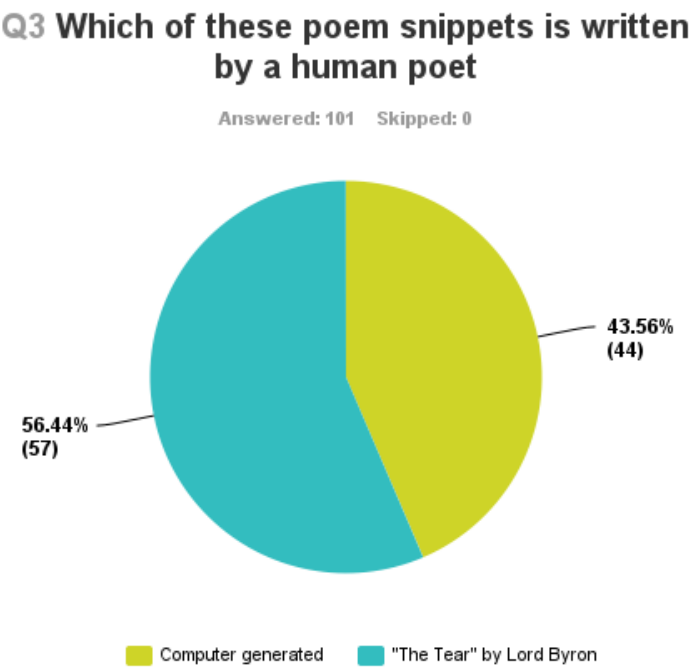

Figure 8.3: Results of Question 3

2. In Figure 8.2, we see, that the computer generated poem was rated first in only 14 % of the cases and better than atleast one of the amateur written poems in 40% of the cases. This shows the subjective nature of poems, because even though the computer generated poem, is now tested against poems written by amateurs, it fares worse than against those written by professionals. Also, it shows that the system generates poems of varying quality, which holds for any human.

3. In Figure 8.3, 40% of the raters mistakenly identified the computer generated poem as written by a human. This proves exactly how comparable the poetry generated by our system, actually is. If readers, can accept the computer generated poetry, as human poetry against poems by famous poets, then the quality of this poem must atleast be good enough from their point of view.

4. Figure 8.4 presents an even better result of 47% for the computer generated poetry, which is almost half of the reviewers. Again, this proves the point explained above and highly supports the claim of the thesis

5. Figure 8.5 analyzes the different features present in the produced poems.

    - 56% agreed that the first poem has rhythm, while 80% could hear the rhythm in the second one. This shows that the rhythm feature has been achieved. What needs to be noted is, that rhythm cannot be always heard when reading

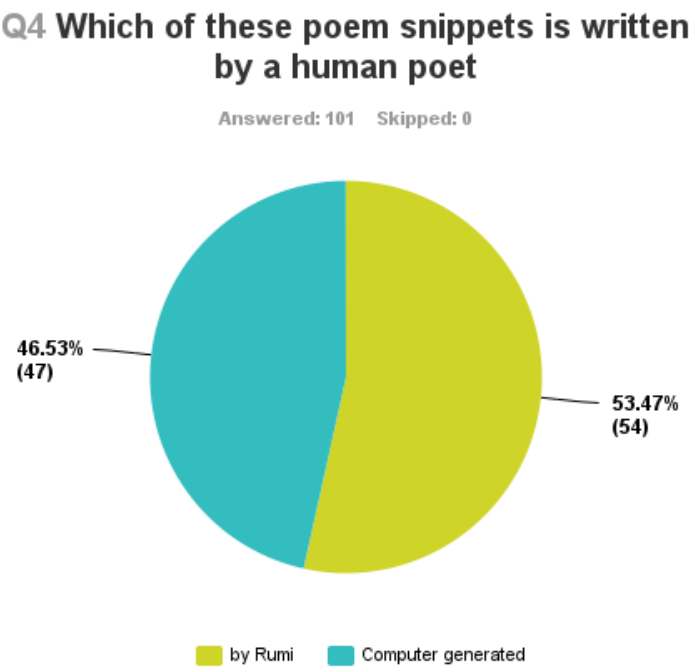

Figure 8.4: Results of Question 4

a poem, because it depend on the pronunciation and language of the reader. Also, most readers do not read the poems out loud, which is usually required to extract the rhythm.

- The first poem has a loose rhyme and thus 70% reviewers said it contained no rhyme. The second poem has a strict rhyme scheme, which almost all the readers agreed with. This shows that the rhymer is accurate.
- 64% of the readers found the message of the first poem clear, while 44% found the second one also clear. These results show that the meaningfulness property has been achieved, to the most part.
- The language of both poems was understandable to an average of 68% of the reviewers, and unclear to around 9%; while an average of 22% found the language odd. This shows that the grammaticality feature was also achieved to the most part, which leads to understandable text. This also tells us that the text has to be adjusted bit, to remove the instances that result in the oddity.
- Finally, the style of the poems was considered somewhat poetic by a total of 54%, and very poetic by 31%. This shows, that 85% thought the poems satisfy the poeticness feature, and more than a quarter of the reviewers found it of high poetic quality, compared to the 15% that did not find the poeticness feature. This and the results of the first two features, show that the poeticness

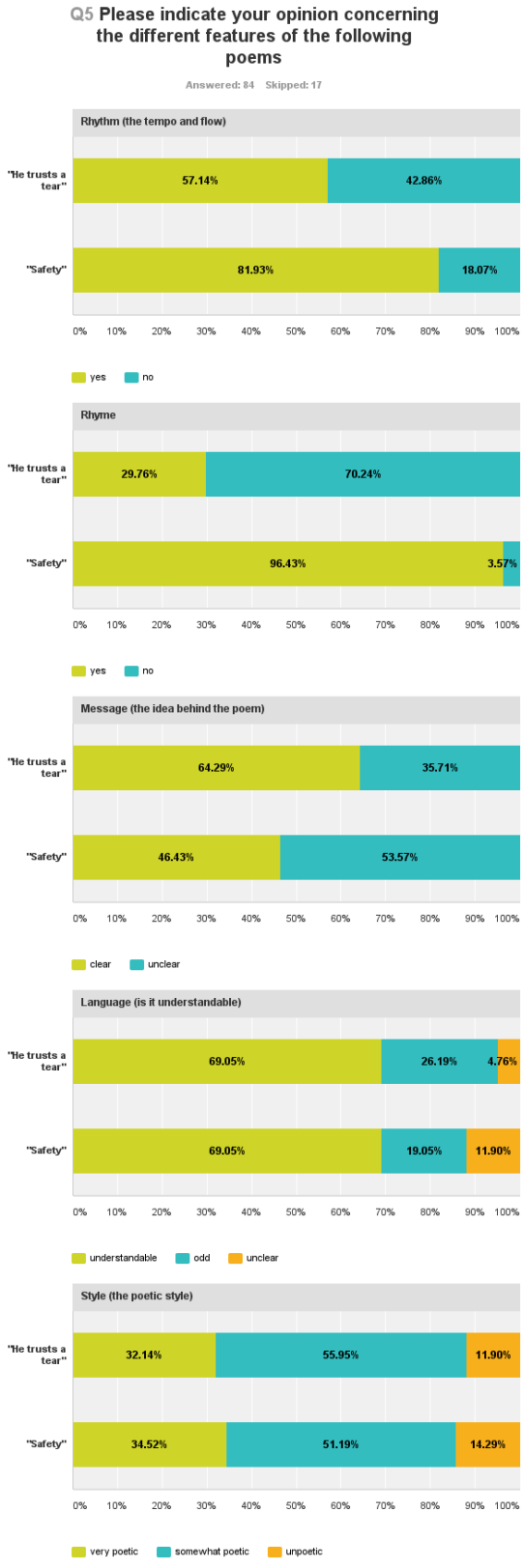

Figure 8.5: Results of Question 5

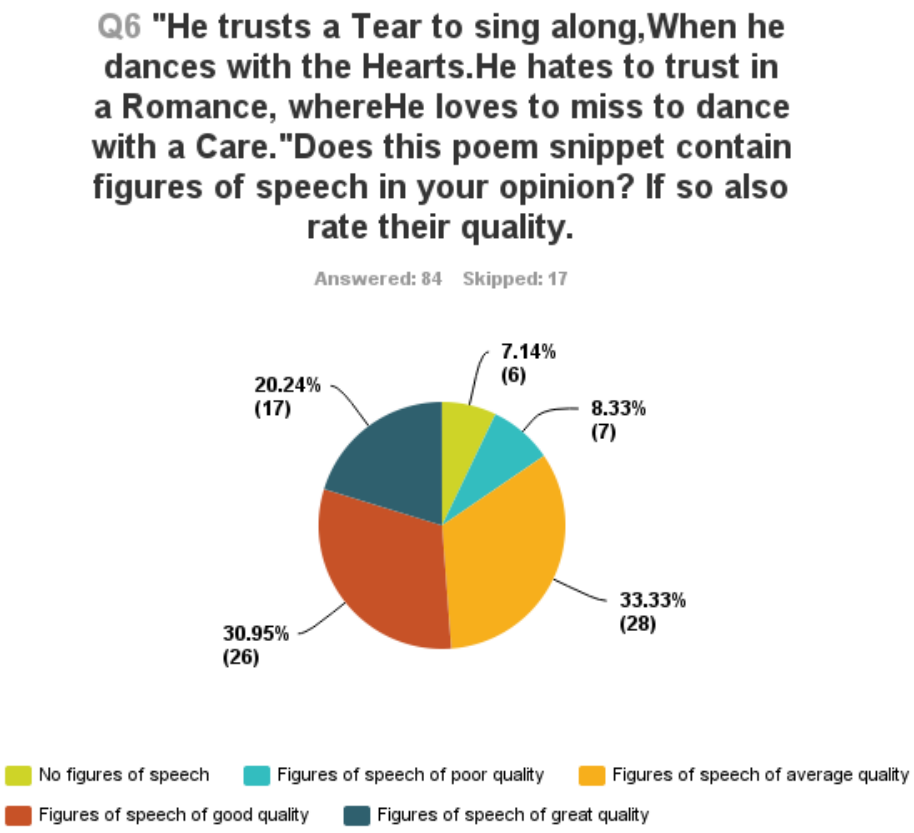

Figure 8.6: Results of Question 6

feature has also been satisfied, to the most part.

6. From Figure 8.6, we can extrapolate, that the claim of achievement of figurative language and metaphors through underspecification has been satisfied. This is so, because only 8% considered the poem to be lacking any figures of speech, while the remaining 92% all found the poem's language figurative. Additionally, around 50% of the readers considered the resulting figures of speech to be of good and even great quality (18 %). This shows, that not only figurative language has been achieved, but that it is of acceptable good quality.

To sum up the results of the survey, we can say that they support our claim, that the developed system, generates poems satisfying the three properties of grammaticality, meaningfulness and poeticness. It also proved, that the generated poems are comparable to, and can match, those written by humans; with different experience and skill levels.

8.2 Comparison

In the following, we will briefly present the poetry generated by only the most relevant poetry generators discussed in section 3, and compare it to the poetry generated by the system developed throughout this thesis.

The following is a snippet of a translated poem, generated by the final version of the system presented in Gervás (2013). It shows that the produced poems are not always grammatically correct, meaningful or coherent, in contrast to the poetry generated by our system.

Alas, my index! Oh, yellow
lemon!....
Do not even find them, put them away
in two little boxed, brother, as if
for white girls.

The system represented in Manurung et al. (2012) will be discussed next. While McGonnagall produces poems, that are very constrained and grammatically correct, it has the disadvantage of being a knowledge-intensive approach, while at the same time relying on a small lexicon, in comparison to the work presented in this paper. McGonnagall also aspires to create poems that are very similar in form and content to existing poems, because it maximizes semantic similarity. This causes the poems to sound too repetitive and lack room for interpretation, as shown in the example:

A very african lion,
who is african, dwells in a waste.
Its head, that is big,
is very big.
A waist, that is its waist, it is small.

Finally, we will consider the poems generated in Toivanen et al. (2013). From the following example, we can see that poems, which are not very meaningful, coherent or grammatically correct can be generated by the system:

Music swells, accent practises, theatre hears!
Her delighted epiphanies bent in her universe:
And then, singing directly a universe she disappears!
An anthem in the judgments after verse!

While the three existing systems rely on information from existing poetry corpora, our system does not depend on any external information, other than the lexicon, in order to generate poetry. This results in unique, entirely artificially generated poems, unlike the ones produced by existing systems, that take their basis and essence from human made poetry. At the same time, the resulting poetry has proven to be comparable to human written poetry, without relying on it.

8.3 Web Application

The final method of presentation and evaluation of the poetry generation system and the achieved results, is the development of a web application for the poetry generation and the resulting poetryThe web application, allows the user to specify the properties of the requested poems and is then provided with a computer generated poem. The user can choose the theme of the poem, as well as its form, rhythm pattern, rhyme scheme and acting persons. The list of user defined options can be extended at any given time. Other than getting real-time poems, the user can also access a repository of already generated poems.Additionally, the user can get an overview of the approach used for poetry generation, as well as accessing the used lexicon. The user can suggest modifications and alterations to the lexicon. Later, this should be extended with a serious game, that would allow the user to help in the lexicon generation.

Finally, the user can choose to help evaluate the generated poetry by following the link for the survey, or by leaving his or her comments and suggestions.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this thesis, a hybrid approach, for generating poems satisfying the three properties of poeticness, grammaticality and meaningfulness, was developed and system following said approach was implemented. The approach does not require existing corpora or external information to generate acceptable poems, which ensures that the produced poems are unpredictable and unique, unlike other approaches that rely on corpora of existing poems. The poetry generation process relies solely on a specially designed lexicon (and the derived sub-lexica) and the reasoner rules written in CHR. This makes it a suitable model of the process of human poetry writing, as humans do not rely on corpora or other poems in order to write new poems, but only on their vocabulary (lexicon), knowledge of the language rules and talent (program code). The generated lexicon provides the reasoner with all the necessary information and contains a large number of words, which is not the case in most of the poetry generation systems. Rather than generating the whole poem text and revising it against the poetry constraints after each iteration like most systems do, the system updates the information for choosing the following words through the syntax and metre pattern, each time a new word is to be added to the poem. This is done to ensure correctness in advance and minimize backtracking. Thus, at any given step all constraints for grammaticality, poeticness and meaningfulness are satisfied simultaneously by pruning the list of possible candidates to fulfill the constraints. An evaluation of the poetry generated by the system, showed that it is comparable to human written poetry. This thesis proves that CHR can be used to develop a hybrid system capable of generating good poetry matching that of humans.

9.2 Future Work

As this thesis deals with natural language generation of creative poetic texts, there are always extensions that can be added towards improving the quality of the produced poems

and widening the scope of the work. The process of generating the main lexicon and the theme-based lexica could be fully automated. Open-Callais Reuters (2014) can be used to generate the sub-lexica. The lexicon could also be extended to include information about the gender of nouns, to enable interchanging them with their corresponding pronouns, when needed. Also, the differentiation between different adverb types could allow for their better placement, resulting in poems that are more grammatically correct. The semantic relation function could be defined for each sentence independently, to improve the coherence and semantics of the poem, while maintaining another value for the semantic relatedness of the whole poem. The value of semantic coherence and meaningfulness could be calculated for each generated poem. The system could incorporate a more sophisticated system for handling figurative language and generating metaphors. Because the system is lexicon-based, it is easily portable to other languages, besides English with, minor changes, given a similar lexicon of the target language. Another possible approach, is the use of CHR Grammars (CHRG) Christiansen (2004) to define a grammar for the sentences of the poem, to handle the grammaticality instead of the used approach. The results of the two approaches should be compared, to choose the superior one. Also, the grammar structure and style of the poem could be learned from existing authors, to reproduce similar poems according to the request of the users. This can be done similar to the approach used in Sneyers and De Schreye (2010). The reasoner could be optimized further to improve the efficiency of the poetry generation system. A more extensive evaluation could be performed. One option would be the anonymous, online posting of the generated poetry in random poetry websites, to get the feedback of a poetry-familiar audience. Also a web application, for generating poetry, analyzing and evaluating its results, should be hosted. This would be beneficial in gathering data and improving the results. Towards this end, the poetry generation system could become more interactive, in the form of a serious game, which would allow the user to manipulate the resulting poetry, to help the system learn and gather data, to improve its future performance.

Bibliography

Alan Beale. Unofficial alternate 12dicts package (alt12dicts). 2014. URL http://aspell.sourceforge.net/wl/.

Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic music composition using answer set programming. *CoRR*, 2010.

Donna M. Campbell. Poetry terms: Brief definitions. 2014. URL http://public.wsu.edu/~campbelld/amlit/poeterms.htm.

Henning Christiansen. CHR grammars. *CoRR*, cs.CL/0408027, 2004. URL http://arxiv.org/abs/cs.CL/0408027.

Simon Colton, Jacob Goodwin, and Tony Veale. Full face poetry generation. In *Proceedings of the Third International Conference on Computational Creativity*, pages 95–102, 2012.

Belén Díaz-Agudo, Pablo Gervás, Pedro Antonio González-Calero, S Craw, and A Preece. Poetry generation in colibri. In *Proceedings of the 6th European Conference on Case Based Reasoning*, Aberdeen, Scotland, 2002. URL http://nil.fdi.ucm.es/sites/default/files/DiazetalECCBR2002.pdf.

Agirrezabal et al. Pos-tag based poetry generation with wordnet. 2013.

Poetry Foundation. Learning lab. 2014. URL http://www.poetryfoundation.org/learning/.

Thom Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1-3):95–138, 1998.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

Pablo Gervás. Wasp: Evaluation of different strategies for the automatic generation of spanish verse. In *Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, University of Birmingham, England, 2000 2000a. URL http://nil.fdi.ucm.es/sites/default/files/GervasAISB2000.pdf.

Pablo Gervás. An expert system for the composition of formal spanish poetry. *JOURNAL OF KNOWLEDGE-BASED SYSTEMS*, 14:200–1, 2000b.

Pablo Gervás. Exploring quantitative evaluations of the creativity of automatic poets. In *Proc. of the 2nd Workshop on Creative Systems, Approaches to Creativity in Artificial Intelligence and Cognitive Science, the 15th European Conf. on Artificial Intelligence (ECAI 2002)*, 2002.

Pablo Gervás. Computational modelling of poetry generation. In *Artificial Intelligence and Poetry Symposium, AISB Convention 2013*, University of Exeter, United Kingdom, 2013.

Pablo Gervás, Dep Sistemas, and Informaticos Programacion. Automatic generation of poetry using a cbr approach. In *In CAEPIA - TTIA 01 Actas Volumen I. CAEPIA*, 2001.

Jerry R. Hobbs. Metaphor, metaphor schemata, and selective inferencing. Technical Report 204, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Dec 1979.

Ray Kurzweil. Ray kurzweil's cybernetic poet. 2001. URL http://www.kurzweilcyberart.com/poetry.

S.R. LEVIN. *Linguistic Structures in Poetry*. 1962. URL http://books.google.com.eg/books?id=3ddAb1_x6aoC.

Robert P Levy. A computational model of poetic creativity with neural network as measure of adaptive fitness. In *Proceedings of the ICCBR-01 Workshop on Creative Systems*. Citeseer, 2001.

Bonacia Ltd. Young writers. 2014. URL https://www.youngwriters.co.uk/index.

Hisar Manurung. An evolutionary algorithm approach to poetry generation. 2004.

Hisar Manurung, Graeme Ritchie, and Henry Thompson. A flexible integrated architecture for generating poetic texts. Technical report, The University of Edinburgh, 2000a.

Hisar Maruli Manurung. Chart generation of rhythmpatterned text. In *Proc. of the First International Workshop on Literature in Cognition and Computers*, 1999.

Hisar Maruli Manurung, Graeme Ritchie, and Henry Thompson. 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, 2000b.

Ruli Manurung, Graeme Ritchie, and Henry Thompson. Using genetic algorithms to create meaningful poetic text. *J. Exp. Theor. Artif. Intell.*, 24(1):43–64, 2012.

Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (2Nd, Extended Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.

Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

H Oliveira. Automatic generation of poetry: an overview. *Universidade de Coimbra*, 2009.

Hugo Gonçalo Oliveira, F Amilcar Cardoso, and Francisco C Pereira. Exploring different strategies for the automatic generation of song lyrics with tra-la-lyrics. In *Proceedings of 13th Portuguese Conference on Artificial Intelligence, EPIA*, pages 57–68, 2007a.

Hugo Gonçalo Oliveira, F Amılcar Cardoso, and Francisco Câmara Pereira. Tra-la-lyrics: An approach to generate text based on rhythm. In *Proceedings of 4th International Joint Workshop on Computational Creativity*, pages 47–55, 2007b.

Thomson Reuters. Open callais. 2014. URL http://www.opencalais.com/.

Lussonnal P Rubaud J and Braffort P. Alamo: Atelier de littérature assisté par la mathématique et les ordinateurs. 2000. URL http://indy.culture.fr/alamo/rialt/pagaccalam.html.

Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1):181–234, 2002.

Jon Sneyers and Danny De Schreye. Apopcaleaps: Automatic music generation with chrism. In *11th International Society for Music Information Retrieval Conference (ISMIR 2010), Utrecht, The Netherlands (August 2010) Submitted*, 2010.

Jukka M Toivanen, Matti Järvisalo, and Hannu Toivonen. Harnessing constraint programming for poetry composition. In *Proceedings of the Fourth International Conference on Computational Creativity*, page 160, 2013.

Jukka Toivanen et al. Corpus-based generation of content and form in poetry. In *Proceedings of the Third International Conference on Computational Creativity*, 2012.

Carnegie Mellon University. The cmu pronouncing dictionary. 2014. URL http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

Princeton University. About wordnet. 2010. URL http://wordnet.princeton.edu.

Sarah Witzig. Accessing wordnet from prolog. Artificial Intelligence Center, The University of Georgia, 2003. URL http://www.ai.uga.edu.

M Tsan Wong and A Hon Wai Chun. Automatic haiku generation using vsm. In *Proceeding of 7th WSEAS International Conference on Applied Computer & Applied Computational Science*. World Scientific and Engineering Academy and Society, 2008.