Harnessing Constraint Programming for Poetry Composition

Jukka M. Toivanen and **Matti Järvisalo** and **Hannu Toivonen**
HIIT and Department of Computer Science
University of Helsinki
Finland

Abstract

Constraints are a major factor shaping the conceptual space of many areas of creativity. We propose to use constraint programming techniques and off-the-shelf constraint solvers in the creative task of poetry writing. We show how many aspects essential in different poetical forms, and partially even in the level of language syntax and semantics can be represented as interacting constraints.

The proposed architecture has two main components. One takes input or inspiration from the user or the environment, and based on it generates a specification of the space and aesthetic of a poem as a set of declarative constraints. The other component explores the specified space using a constraint solver.

We provide an elementary set of constraints for composition of poetry, we illustrate their use, and we provide examples of poems generated with different sets of constraints.

Introduction

Rules and constraints can be seen as an essential ingredient of creativity. First, there typically are strong constraints on the creative artefacts. For instance, consider traditional western music. In order for a composition to be recognized as (western) music in the first place, it must meet a number of requirements concerning, e.g., timbre, scale, melody, harmony, and rhythm. For any specific genre of western music, the constraints usually become much tighter.

Similarly, the composition of many types of poetry is governed by numerous rules specifying such things as strict stress and syllable patterns, rhyming and alliteration structures, and selection of words with certain associations — in addition to the basic constraints of syntax and semantics that are needed to make the expressions understandable and meaningful.

However, constraints are not just a nuisance that creative agents need to cope with in order to produce plausible results. On the contrary, constraints are often considered to be an essential source of creativity for humans. For instance, composer Igor Stravinsky associated constraints with creating freedom, not containment:

"The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit."

(Stravinsky 1947)

Constraints can also be used as computational tools for studies of creativity or creative artefacts. Artificial intelligence researcher Marvin Minsky suggested that a good way to learn about how music "worked" was to represent musical compositions as interacting constraints, then modify these constraints and study their effects on the musical structures (Roads 1980). This essential idea has been explored extensively in the field of computer music research afterwards.

Our domain of interest in this paper is composition of poetry. We envision a computational environment where formally expressed constraints and constraint programming methods are used to (1) specify a conceptual search space, (2) define an aesthetic of concepts in the space, (3) explore the space to find the most aesthetic concepts in it.

Any given set of (hard) constraints on poems specifies a space of possible poems. For instance, the number of lines and the number of syllables per line could be such constaints, contributing to the style of poetry. Soft constraints, in turn, can be used to indicate (aesthetical) preferences over poems and to rank poems that match the hard constraints. For instance, rhyme could be a soft constaint, giving preference to poems that follow a given rhyme structure but not absolutely requiring it.

In this paper we study and illustrate the power of constraint programming for creating poems. In our current setup, the creative system consists of two subcomponents. One takes input from user or from some other source of inspiration, and based on it *specifies* the space and poetical aesthetic (as a set of constraints). The other subcomponent *explores* the specified space using the aesthetic, i.e., produces optimally aesthetic poems in the space (using a constraint solver).

We show how poems can be generated by applying different kinds of constraints and constraint combinations using an off-the-shelf constraint programming tool. The elegance of this approach is that it is not based on specifying a step sequence to produce a certain kind of a poem, but rather on declaring the properties of a solution to be found using mathematical constraints. An empirical evaluation of the obtained poetry is left for future work.

We next briefly review some related work on constraint programming in creative applications, and on poetry generation. Then we provide a description of a constraint model for composing poems, illustrating the ideas with examples. We discuss the results and conclude by outlining future work.

Related Work

Constraint-based methods have been applied in various fields such as configuration and verification, planning, and evolution of language, to name a few. In the area of computational creativity, constraints have been used mostly to describe the composition of various aspects of music. For example, Boenn et al. (2011) have developed an extensive music composition system called Anton which uses Answer Set Programming to represent the musical knowledge and the rules of the system. Anton describes a model of musical composition as a collection of interacting constraints. The system can be used to compose short pieces of music as well as to assist the composer by making suggestions, completions, and verifications to aid in the music composition process.

On the other hand, composition of poetry with constraint programming techniques has received little if any attention. Several different approaches have been used (Manurung, Ritchie, and Thompson 2000; Gervás 2001; Manurung 2003; Diaz-Agudo, Gervás, and González-Calero 2002; Wong and Chun 2008; Netzer et al. 2009; Colton, Goodwin, and Veale 2012; Toivanen et al. 2012), many involving constraints in one form or another, but we are not aware of any other work systematically based on constraints and implemented using a constraint solver.

The system developed by Manurung et al. (2003) uses a grammar-driven formulation to generate metrically constrained poetry out of a given topic. This approach performs stochastic hillclimbing search within an explicit state-space, moving from one solution to another. The explicit representation is based on a hand-crafted transition system. In contrast, we employ constraint-programming methodology based on searching for optimal solutions over an implicit representation of the conceptual space. Our approach should scale better to large numbers of constraints and a large input vocabulary than explicit state-space search.

The ASPERA poetry composition system (Gervás 2001), on the other hand, uses a case-based reasoning approach. This system generates poetry out of a given input text via composition of poetic fragments retrieved from a case-base of existing poetry. These fragments are then combined together by using additional metrical rules.

The Full-FACE poetry generation system (Colton, Goodwin, and Veale 2012) uses a corpus-based approach to generate poetry according to given constraints on, for instance, meter and stress. The system is also argued to invent its own aesthetics and framings of its work. In contrast to our system, this approach uses constraints to shape only some aspects of the poetry composition procedure whereas our approach is fully based on expressing various aspects of poetry as mutually interacting constraints and using a constraint-solver to efficiently search for solutions.

The approach of this paper extends and complements our previous work (Toivanen et al. 2012). We proposed a method where a template is extracted randomly from a given corpus, and words in the template are substituted by words related to a given topic. Here we show how such basic functionality can be expressed with constraints, and more interestingly, how constraint programming can be used to add control for rhyme, meter, and other effects.

Simpler poetry generation methods have been proposed, as well. In particular, Markov chains have been widely used to compose poetry. They provide a clear and simple way to model some syntactic and semantic characteristics of language (Langkilde and Knight 1998). However, the resulting poetry tends to have rather poor sentence and poem structures due to only local syntax and semantics.

Overview

The proposed poetry composition system has two subcomponents: a conceptual space *specifier* and a conceptual space *explorer*. The former one determines what poems can be like and what kind of poems are preferred, while the latter one assumes the task of producing such poems.

The modularity and the explicit specification of the conceptual search space have great potential benefits. Modularity allows one to (partially) separate the content and form of poetry from the computation needed to produce matching poetry. An explicit, declarative specification, in turn, gives the creative system a potential to introspect and modify its own goals and intermediate results (a topic to which we will return in the conclusion).

A high-level view to the internal structure of the poetry composition system considered in this work is shown in Figure 1. In this paper, our focus is on the explorer component and on the interface between the components. Our specifier component is built on the principles of Toivanen et al. (2012), but ideas from many other poetry generation systems (Gervás 2001; Manurung 2003; Colton, Goodwin, and Veale 2012) could be used in the specifier component as well.

The assumption in the model presented here is that the specifier can generate a large number of mutually dependent choices of words for different positions in the poem, as well as dependencies between them. The specifier uses input from the user and potentially other sources as its inspiration and parameters and automatically generates the input for the explorer component, shielding user from the details of constraint programming.

The automatically generated "data" or "facts" are conveyed to the explorer component that consists of a constraint solver and a static library of constraints. The library is provided by the system designers, i.e., by us, and any constraints that the specifier component wishes to use are triggered by the data it generates. The user of the system does not need to interact directly with the constraint library (but the specifier component may offer the user options for choosing which constraints to use).

Our focus in this paper is on the explorer component, and in the constraint specifications that it receives from the specifier component or from the static library:

- The number of lines, and the number of words on each line (we call this the *skeleton* of the poem).

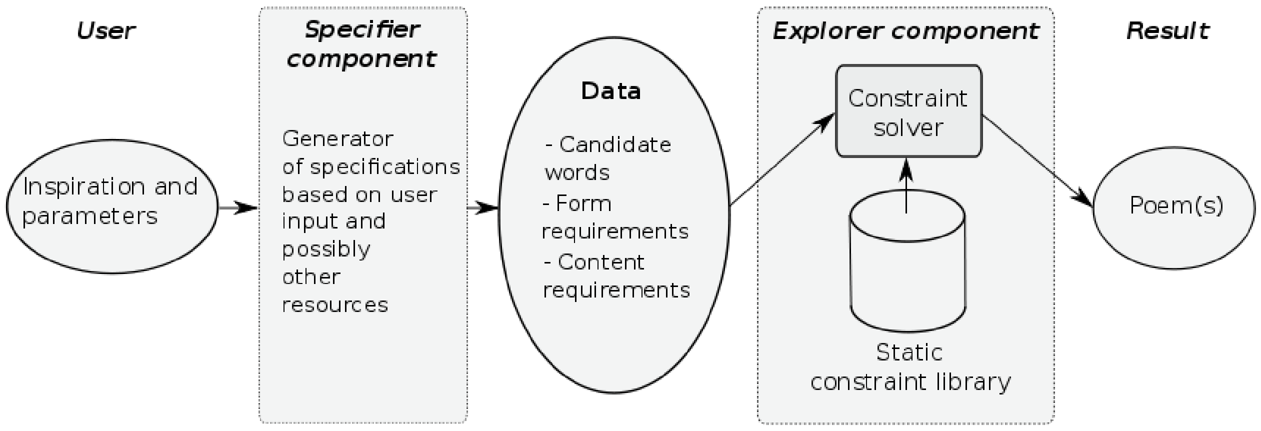

Figure 1: Overview of the poetry composition workflow. The user provides some inspiration and parameters, based on which the space specifier component generates a set of constraints, used as "data" by the constraint solver in the explorer component. The explorer component additionally contains a static library of constraints that are dynamically triggered by the data. Explorer component then outputs a poem that best fulfills wishes of the user.

- For each word position in the skeleton, a list of words that potentially can be used in the position (collectively called the *candidates*).
- Possible additional requirements on the desired form of the poem (e.g., rhyming structure).
- Possible additional requirements on the syntax and contents of the poem (e.g., interdependencies between words to make valid expressions).

We will next describe these in more detail.

Poetry Composition via Answer Set Programming

The explorer component takes as input specifications dynamically generated by the specifier, affecting both the search space and the aesthetic. In addition, it uses a static constraint library. Together, the dynamic specifications and the constraint library form a constraint satisfaction problem (or, by extension, an optimization problem; see end of the section). The constraint satisfaction problem is built so that the solutions to the problem are in one-to-one correspondence with the poems that satisfy the requirements imposed by the specifier component of the system (as potentially instructed by the user). Highly optimized off-the-shelf constraint satisfaction solvers can then be used to find the solutions, i.e., to produce poems.

In this work, we employ *answer set programming* (ASP) (Gelfond and Lifschitz 1988; Niemelä 1999; Simons, Niemelä, and Soininen 2002) as the constraint programming paradigm, since ASP allows for expressing the poem construction task in an intuitively appealing way. At the same time, state-of-the-art ASP solvers, such as Clasp (Gebser, Kaufmann, and Schaub 2012), provide an efficient way of finding solutions to the poem construction task. Furthermore, ASP offers in-built support for constraint optimization, which allows for searching for a poem of high quality with respect to different imposed quality measures.

We will not provide formal details on answer set programming and its underlying semantics; the interested reader is referred to other sources (Gelfond and Lifschitz 1988; Niemelä 1999; Simons, Niemelä, and Soininen 2002) for a detailed account. Instead, we will in the following provide a step-by-step intuitive explanation on how the task of poetry generation can be expressed in the language of ASP. For more hands-on examples on how to express different computational problems in ASP, we refer the interested reader to Gebser et al. (2008).

Answer set programming can be viewed as a data-centric constraint satisfaction paradigm, in which the input data, represented via *predicates*, expresses the problem instance. In our case, this dynamically generated data will express, for example, basic information on the poem skeleton (such as length of lines), and the candidate words within the input vocabulary that can be used in different positions within the poem. The actual computational problem (in our case poetry generation) is expressed via *rule-based constraints* which are used for inferring additional knowledge based on the input data, as well as for imposing constraints over the solutions of interest. The rule-based constraints constitute the static constraint library: once written, they can be reused in any instances of poem generators just by generating data that activates the constraints. Elementary constraints are an integral part of the system — comparable to program code. More rule-based constraints can be added by the specifier component if needed. The end-user does not need to write any constraints.

The Basic Model

We next describe a constraint library, starting with elementary constraints. We also illustrate dynamically generated specifications. While these are already sufficient to generated poetry comparable to that of Toivanen et al. (2012), we remind the reader that these constraints are examples illustrating the flexibility of constraint programming in compu-

Table 1: The predicates used in the basic ASP model

Predicate	Interpretation
`rows(X)`	the poem has `X` rows
`positions(X,Y)`	the poem contains `Y` words on row `X`
`candidate(W,I,J,S)`	the word `W`, containing `S` syllables, is a candidate for the `J`th word of the `I`th line
`word(W,I,J,S)`	the word `W`, containing `S` syllables, is at position `J` on row `I` in the generated poem

```
% Generator part
{ word(W,I,J,S) } :- candidate(W,I,J,S).                              (G1)

% Testing part: the constraints
:- not 1 { word(W,I,J,S) } 1, rows(X), I = 1..X, positions(I,Y), J=1..Y. (T1)
```

Figure 2: Answer set program for generating poetry: the basic model

tational poetry composition, and different sets of constraints can be used for different effects.

We will first give a two-line basic model of the constraint library that takes the skeleton and candidates as input. This model simply states that exactly one of the given candidate words must be selected for each word position of the poem.

Predicates The predicates used in the basic answer set program are listed in Table 1, together with their intuitive interpretations.

The input predicates `rows/1` and `positions/2` characterize the number of rows and the number of words allowed on the individual rows of the generated poems. The input predicate `candidate/4` represents the input vocabulary, i.e., the words that may be considered as candidates for words at specific positions.

The output predicate `word/4` represents the solutions to the answer set program, i.e., the individual words and their positions in the generated poem.

Example. The following is an example of the basic structure of a data file representing a possible input to the basic ASP model

```
rows(6).
positions(1,6).
positions(2,8).
positions(3,8).
positions(4,5).
positions(5,6).
positions(6,6).

candidate("I",1,1,1).
candidate("melt",1,2,1).
candidate("weed",1,2,1).
candidate("teem",1,2,1).
candidate("kidnap",1,2,2).
candidate("perspire",1,2,2).
candidate("shut",1,2,1).
candidate("eclipse",1,2,1).
candidate("sea",1,2,1).
candidate("plan",1,2,1).
candidate("hang",1,2,1).
candidate("police",1,2,2).
candidate("revamp",1,2,2).
candidate("flip",1,2,1).
candidate("wring",1,2,1).
candidate("sting",2,2,2).

...
```

Rules The answer set program that serves as our basic model for generating poetry is shown in Figure 2. The program can be viewed in two parts: the *generator part* (Rule G1) and the *testing part* (Rule T1). The test part consists of rule-based constraints that filter out poems that do not satisfy the conditions for acceptable poems characterized by the program.

In the generator part, Rule G1 states that each candidate word for a specific position of the poem may be considered to be chosen as the word at that position in the generated poem (expressed using the so-called *choice* construct { `word(W,I,J,S)` }).

In the testing part, Rule T1 imposes the most fundamental constraint that exactly one candidate word should be chosen for each word position in the poem: the empty left-hand-side of the rule is interpreted as *falsum*, a contradiction. The rule then states that, for each row and each position on the row, it is a contradiction if it is *not* the case that exactly one word is chosen for that position (expressed as the *cardinality* construct `1` { `word(W,I,J,S)` } `1`).

Example. Given the data presented above these basic rules are now grounded as follows. There are six lines in the poem as described by the *rows* predicate and each of these lines has a certain number of positions to be filled with words as described by the *positions* predicate. The *candidate* predicates specify which words are suitable choices for these positions. During grounding the solver tries to find a suitable candidate for each position, which is trivial in the basic model that lacks any constraints between the words. We consider more interesting models next.

Controlling the Form of Poems

We will now describe examples of how the form of the poems being generated can be further controlled in a modular fashion by introducing additional predicates and rules over these predicates to the basic ASP model. The additional predicates introduced for these examples are listed in

Table 2: Predicates used in extending the basic ASP model

Predicate	Interpretation
`must_rhyme(I,J,K,L)`	the word at position `J` on row `I` and the word at position `L` on row `K` are required to rhyme
`rhymes(X,Y)`	the words `X` and `Y` rhyme
`nof_syllables(I,C)`	the `I`th row of the poem is required to contain `C` syllables
`min_occ(W,L)`	`L` is the lower bound on the number of occurrence of the word `W`
`max_occ(W,U)`	`U` is the upper bound on the number of occurrence of the word `W`

```
% Generator part

{ word(W,I,J,S) } :- candidate(W,I,J,S).                              (G1)
rhymes(Y,X) :- rhymes(X,Y).                                           (G2)
syllables(W,S) :- candidate(W,_,_,S).                                 (G3)

% Testing part: the constraints

:- not 1 { word(W,I,J,S) } 1, rows(X), I = 1..X, positions(I,Y), J=1..Y. (T1)
:- word(W,I,J,S), word(V,K,L,Q), must_rhyme(I,J,K,L), not rhymes(W,V).   (T2)
:- Sum = #sum [ word(W,I,J,S) = S ], Sum != C, nof_syllables(I,C),      (T3)
   I = 1..X, rows(X).
:- not L { word(W,_,_,_) } U, min_occ(W,L), max_occ(W,U).                (T4)
```

Figure 3: Answer set program for generating poetry: extending the basic model

Table 2. Using these predicates, rules that refine the basic model are shown in Figure 3 (Rules G2, G3, and T2–T4).

Rhyming The predicate `must_rhyme/4` is used for providing pairwise word positions that should rhyme. Knowledge on the pairwise relations of the candidate words, namely, which pairs of candidate words rhyme, is provided via the `rhymes/2` predicate. Rule G2 enforces that rhyming of two words is a symmetry relation. In the testing part Rule T2 imposes the constraint that, in case two words chosen for specific positions in a poem must rhyme, but the chosen two words do not rhyme, a contradiction is reached.

Numbers of Syllables The basic model can also be extended to generate poetical structures with more specific constraints. As an example, one can consider forms of poetry that have strict constraints on the numbers of syllables in every line, such as haikus, tankas, and sonnets.

We use the additional predicate `nof_syllables/2` for providing as input the required number of syllables on the individual rows. At the same time, Rule G3 projects the information on the number of syllables of each candidate word to the `syllables/2` predicate. Rule T3 can then be used to ensure that the number of syllables on each row (line) of the poem (computed and stored in the `Sum` variables using the counting construct `Sum = #sum [ word(W,I,J,S) = S ]`) matches the number of syllables specified for the row by the `nof_syllables/2` predicate.

Word Occurrences The simple model above does not control possible repetitions of words at all. Such control can be easily added by introducing input predicates `min_occ(W,L)` and `max_occ(W,U)`, which are then used to state for each word `W` the minimum `L` (respectively, maximum `U`) number of occurrences allowed for the word. Using these additional predicates, Rule T4 then constrains the number of occurrences to be within these lower and upper bounds (expressed by the cardinality constraint `L { word(W,_,_,_) } U`).

Further Possibilities for Controlling Form The possibilities of controlling poetical forms are not of course limited to simple requirements for fulfilment of certain syllable structures or rules for rhyming and alliteration. Besides strict constraints on numbers of syllables on verse, classical forms of poetry usually obey a specific stress pattern, as well. Stress can be handled with constraints similar to the ones governing syllables. Metric feet like iamb, anapest, and trochee can be used by specifying constraints that describe positions where the syllable stress must lie in every line of verse.

Controlling poetical form also provides interesting possibilities for using constraint optimization techniques (to be described below). As an example, consider different forms of lipograms i.e. poems that avoid a particular letter like *e* or univocal poems where the set of possible vowels in the poem is restricted to only one vowel. Similarly, more complex optimisations of the speech sound structure can be handled depending on whether the wished poetry is required to have soft or brutal sound, or to have characteristics of a tongue-twister.

Controlling the Contents and Syntax of Poems

While the example constraints presented above focus on controlling the form of poems, linguistic knowledge of phonology, morphology, and syntax (as examples) can similarly be controlled by introducing additional constraints in a modular fashion. This includes rules of syntax that specify

```
failed_rhyme(I,J,K,L) :- word(W,I,J,S), word(V,K,L,Q),
                         must_rhyme(I,J,K,L), not rhymes(W,V).          (T2')
failed_syllable_count(I) :- Sum = #sum [ word(W,I,J,S) = S ], Sum != C,
                            nof_syllables(I,C), I = 1..X, rows(X).      (T3')
failed_occount(W) :- not L { word(W,_,_,_) } U, min_occ(W,L), max_occ(W,U). (T4')

#minimize [ failed_rhyme(I,J,K,L) @3 ].                                 (O2)
#minimize [ failed_syllable_count(I) : I=1..X : rows(X) @2 ].           (O3)
#minimize [ failed_occount(W) @1 ].                                     (O4)
```

Figure 4: Handling inconsistencies by relaxing the constraints and introducing optimization criteria

how linguistic elements are sequenced to form valid statements and rules of semantics which specify how valid references are made to concepts.

Consider, for example, transitive and intransitive verbs, i.e., verbs that either require or do not require an object to be present in the same sentence. Here one can impose additional constraints for declaring which words can or cannot be used in the same sentence where a transitive verb requiring certain preposition and an object has been used. Similarly other constraints not directly related to the poetical forms but rather to linguistic structures like idioms, where several words are always bundled together, can be effectively declared as constraints. The same holds for syntactic aspects such as rules governing the constituent structure of sentences (Lierler and Schüller 2012).

As a simple, more concrete example, consider the following. In order to declare that the poems of interest start with the word "I", the fact `word("I",1,1,1).` can be added to the constraint model. In order to ensure that all verbs associated with the first person should be in past tense, the additional predicate `in_past_tense/1` can be introduced, and specified for each past-tense verb in the data. Combining the above, one can as an example declare that the word following any "I" is in a past tense, using the following two rules.

```
:- word("I",I,J,1), word(W,I,J+1,_),
   not in_past_tense(W).
```

```
:- word("I",I,J,1), positions(I,J),
   word(W,I+1,1,_), not in_past_tense(W).
```

Here the first rule handles the case that the occurrence of "I" is not the last word on a row. The second rule handles the case that "I" is the last word on a row, in which case the first word on the following row should be in past tense.

More generally, one can pose constraints that ensure that two (or more) words within a poem are compatible (in some specified sense), even if the words are not next to each other. For an example, consider the additional predicated `pronoun/1` and `verb/1` that hold for words that are pronouns and verbs, respectively, and the predicate `person/2` that specifies the grammatical person, expressed as an integer value, of a given word: `person(W,P)` is true if and only if the word `W` has person `P`. Using these predicates, one can enforce that, for the first verb following any pronoun (not necessarily immediately after the pronoun), the pronoun and the verb have to have the same person. For instance, after the pronoun "she" the first following verb has to be in the third person singular form. This can be expressed as the following rule:

```
:- word(W,I,J,_), pronoun(W), person(W,P),
   0{ word(U,I,L,_) : verb(U) : L>J : L<K }0,
   word(V,I,K,_), verb(V), person(V,Q),
   K>J, P!=Q.
```

Similarly, by specifying the additional predicate `verb/1` for each verb in the input data, one can require that the whole poem should be in past tense:

```
:- word(W,_,_,_), verb(W),
   not in_past_tense(W).
```

Specifying an Aesthetic via Optimization

Up to now, we have only considered hard constraints, and did not address how to assess the aesthetics of generated poems, or how to generate poems that are maximally aesthetic by some measures.

In the constraint programming framework, an aesthetic can be specified using *soft constraints*. The constraint solver then attempts to look for poems which maximally satisfy the soft constraints. In ASP, this is achieved by using *optimization* statements offered by the language.

As concrete examples, we will now explain how Rules T2–T4 can be turned into soft constraints. The soft variants, Rules T2'–T4', are shown in Fig. 4, together with the associated optimization statements O2–O4. Taking Rule T3 as an example, the idea is to introduce a new predicate `failed_syllable_count/1` with the following interpretation: Predicate `failed_syllable_count(I)` is true for row `I` if and only if the number of syllables on the row was not made to match the required number. In contrast to Rule T3, which rules out all solutions of the model immediately in such a case, Rule T3' simply results in assigning `failed_syllable_count(I)` to true. Thus the predicate `failed_syllable_count/1` acts as an indicator of failing to have the required number of syllables on a specific row.

The optimization statement associated with Rule T3' is Rule O3. This `minimize` statement declares that the number of rows `I` for which `failed_syllable_count(I)` is assigned to true should be minimized, or equivalently, that the numbers of syllables should conform to the required numbers of syllables for as many rows as possible. The optimization variants T2' and T4' and the associated optimization statements follow a similar scheme.

When multiple such optimization statements are introduced to the model, the relative importance of the statements is declared using the `@i` attached to each of the optimization statement. In the example of Figure 4, the primary objective is to minimize the number of rhyming failures (specified using `@3`). The secondary objective is then to find, among the set of poems that minimize this primary objective, a poem that has a minimal number of lines with a wrong number of syllables, (using `@2`), and so forth.

Examples

We will now illustrate the results and effects of some combinations of constraints.

In the data generation phase (the specifier component) we use the methodology by Toivanen et al. (2012), including the Stanford POS-tagger and morpha & morphg inflectional morphological analysis and generation tools (Toutanova et al. 2003; Minnen, Carroll, and Pearce 2001). The poem templates are extracted automatically from a corpus of human-written poetry. The only input by the user is a topic for the poem, and some other parameters as described below.

As a test case for our current system we study how the approach manages to produce different types of *quatrains*. It is a unit of four lines of poetry; it may either stand alone or be used as a single stanza within a larger poem. The quatrain is the most common type of stanza found in traditional English poetry, and as such is fertile ground on which to test theories of the rules governing poetry patterns.

The specifier component randomly picks a quatrain from a given corpus of existing poetry. It then automatically analyses its structure, to generate a skeleton for a new poem. The following poem skeleton is marked with the required part-of-speech for every word position (PR = pronoun, VB = verb, PR_PS = possessive pronoun, ADJ = adjective, N_SG = singular noun, N_PL = plural noun, C = conjunction, ADV = adverb, DT = determiner, PRE = preposition):

N_SG VB, N_SG VB, N_SG VB!
PR_PS ADJ N_PL ADJ PRE PR_PS N_SG:
– C ADV, ADV ADV DT N_SG PR VB!
DT N_SG PRE DT N_PL PRE N_SG!

The specifier component then generates a list of candidate words for each position. If we give "music" as the topic of the poem, the specifier specifically uses words related to music as candidates, where possible (Toivanen et al. 2012). A large number of poems are possible, in the absense of other constraints, and the constraint solver in the explorer component outputs this one (or any number of alternative ones, if required):

Music swells, accent practises, traditionalism marches!
Her devote narrations bent in her improvisation:
– And then, vivaciously directly a universe she ventilates!
An anthem in the seasons of radio!

This example does not yet have any specific requirements for the prosodical form. Traditional poetry often has its prosodic structure advertised by one or more of several poetic devices, with rhyming and alliteration being best-known of these. Let the specifier component hence generate the additional constraints that the first and the third line must rhyme, as well as the second and fourth line. As a result of this more constrained specification we now get a very similar poem, but with some words changed to rhyme.

Music swells, accent practises, traditionalism hears!
Her devote narrations bent in her chord:
– And then, vivaciously directly a universe she disappears!
An anthem in the seasons of record!

Addition of this simple constraint adds rhyme to the poem, which in turn draws attention to the prosodic structure of the poem. Use of prosodic techniques to advertise the poetical nature of a given text can also enhance coherence of the poetry as the elements are linked together more tightly. For example, a rhyme scheme of ABAB would give the listener a strong sense that the first and third as well as the second and fourth lines belong together as a group, heightening the saliency of the alternating structure that may be present in the content, as well.

The constraint on rhyming reflects the intuition that rhyme works by creating expectation and satisfaction of that expectation. Upon hearing one line of verse, the listener expects to hear another line that rhymes with it. Once the second rhyme is heard, the expectation is fulfilled, and a sense of closure is achieved. Similarly, adding constraints that specify a more sophisticated prosodic structure or content related aspects may lead to improved quality of the generated poetry.

Let us conclude this section with an example of an aesthetic, an optimization task concerning the prosodic structure of poetry. Consider composition of lipograms, i.e., poems avoiding a particular letter. (Also univocalism or more complex optimizations of the occurrence of certain speech sounds can be composed in a similar fashion.) The following poem is an example of a lipogram that avoids the letter *o*. As a result of this all words that contained that letter in the previous example are changed to match the strengthened constraints:

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 judgements after verse!

Empirical results of Toivanen et al. (2012) indicate that in Finnish, already the basic mechanism produces poems of surprisingly high quality. The sequence of poems above illustrates how their quality can be substantially improved by relatively simple addition of new, declarative constraints.

Discussion and Conclusions

We have proposed harnessing constraint programming for composing poetry automatically and flexibly in different styles and forms. We believe constraint programming has high potential in describing also other creative phenomena.

A key benefit is the declarativity of this approach: the conceptual space is explicitly specified, and so is the aesthetic, and both are decoupled from the algorithm for exploring the search space (an off-the-shelf constraint solver). Due to its modular nature, the presented approach can be an effective building block of more sophisticated poetry generation systems.

An interesting next step for this work is to build an interactive poetry composition system which makes use of constraint programming in an iterative way. In this approach the constraint model is refined and re-solved based on user feedback. This can be seen as an iterative abstract-refinement process, in which the first abstraction specifies a very large search-space that is iteratively pruned by refining the constraint model with more intricate rules that focus search to the most interesting parts of the conceptual space.

Another promising research direction is to consider a self-reflective creative system. Since the search space and aesthetic are expressed in an explicit manner as constraints, they can also be observed and manipulated. We can envision a creative system that controls its own constraints. For instance, after observing that a large amount of good results is obtained with the current constraints, it may decide to add new constraints to manipulate its own internal objectives. Modification of the set of constraints may lead to different conceptual spaces and eventually to transformational creativity (Boden 1992). Development of metaheuristics and learning mechanisms that enable such self-supported behavior is a great challenge indeed.

Acknowledgements

This work has been supported by the Academy of Finland under grants 118653 (JT,HT), and 132812 and 251170 (MJ).

References

Boden, M. 1992. *The Creative Mind.* London: Abacus.

Boenn, G.; Brain, M.; vos, M. D.; and Ffitch, J. 2011. Automatic music composition using answer set programming. *Theory and Practice of Logic Programming* 11(2-3):397–427.

Colton, S.; Goodwin, J.; and Veale, T. 2012. Full-face poetry generation. In *International Conference on Computational Creativity*, 95–102.

Diaz-Agudo, B.; Gervás, P.; and González-Calero, P. A. 2002. Poetry generation in COLIBRI. In *ECCBR 2002, Advances in Case Based Reasoning*, 73–102.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2008. A user's guide to gringo, clasp, clingo, and iclingo. `http://downloads.sourceforge.net/potassco/guide.pdf?use_mirror=`.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187:52–89.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, 1070–1080.

Gervás, P. 2001. An expert system for the composition of formal spanish poetry. *Journal of Knowledge-Based Systems* 14(3–4):181–188.

Langkilde, I., and Knight, K. 1998. The practical value of n-grams in generation. In *Proceedings of the International Natural Language Generation Workshop*, 248–255.

Lierler, Y., and Schüller, P. 2012. Parsing combinatory categorial grammar via planning in answer set programming. In Erdem, E.; Lee, J.; Lierler, Y.; and Pearce, D., eds., *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, 436–453. Springer.

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

Manurung, H. 2003. *An evolutionary algorithm approach to poetry generation*. Ph.D. Dissertation, University of Edinburgh, Edinburgh, United Kingdom.

Minnen, G.; Carroll, J.; and Pearce, D. 2001. Applied morphological processing of English. *Natural Language Engineering* 7(3):207–223.

Netzer, Y.; Gabay, D.; Goldberg, Y.; and Elhadad, M. 2009. Gaiku : Generating haiku with word associations norms. In *Proceedings of NAACL Workshop on Computational Approaches to Linguistic Creativity*, 32–39.

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

Roads, C. 1980. Interview with Marvin Minsky. *Computer Music Journal* 4.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Stravinsky, I. 1947. *Poetics of Music*. Cambridge, MA: Harvard University Press.

Toivanen, J. M.; Toivonen, H.; Valitutti, A.; and Gross, O. 2012. Corpus-based generation of content and form in poetry. In *International Conference on Computational Creativity*, 175–179.

Toutanova, K.; Klein, D.; Manning, C.; and Singer, Y. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL, Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 252–259.

Wong, M. T., and Chun, A. H. W. 2008. Automatic haiku generation using VSM. In *Proceedings of ACACOS, The 7th WSEAS International Conference on Applied Computer and Applied Computational Science*, 318–323.