Generating Code For Expressing Simple Preferences: Moving On From Hardcoding And Randomness

Michael Cook and Simon Colton
Computational Creativity Group
Goldsmiths, University of London
gamesbyangelina.org — ccg.doc.gold.ac.uk

Abstract

Software expressing intent and justifying creative decisions are important considerations when building systems in the context of Computational Creativity. However, getting software to express subjective opinions like simple preferences is difficult without mimicking existing people's opinions or using random choice. In this paper, we propose an alternative way of enabling software to make meaningful decisions in small-scale subjective scenarios, such as choosing a favourite colour. Our system uses a combination of metrics as a fitness function for evolving short pieces of code that choose between artefacts. These 'preference functions' can make choices between simple items that are neither random nor based on an already existing opinion, and additionally have a sense of consistency. We describe the system, offer some example results from the work and suggest how this might lead to further developments in generative subjectivity in the future.

Introduction

Computationally creative software usually makes many decisions in the process of producing an artefact. These decisions are often in the context of problems for which 'notions of optimality are not defined' (Eigenfeldt, Burnett, and Pasquier 2012) and so there is no definitive equation or objective measure that can guide them to the 'best' answer. As a compromise, the developers of such software provide ways to guide the software in making these decisions: sometimes by providing predetermined heuristics; sometimes by allowing the software to create models trained on decisions made by people; sometimes using random chance.

In many of these creative decisions there are no right or wrong answers. For example, in (Veale 2013) a system writes poetry by first generating several potential metaphors to work from. These metaphors are all considered good candidates that could produce poems – the system selects one at random, because it has no meaningful reason to choose between them. This is a small decision within a much larger system, and in many ways it is insignificant compared to the larger creative act the software performs. In this paper, however, we argue that there are two important consequences to relying on random choice or predetermined heuristics for decisions such as this. Firstly, we prevent our software from intelligently discussing these choices in framing information, and as a result miss out on opportunities to add value to the artefacts created or raise the perception of our software as creative (Colton, Charnley, and Pease 2011). Secondly, when observers discover or are informed that these choices are made due to external factors or randomness, then we contend that their perception of the software as creative is lowered significantly, even if the decision in question seems trivial.

Software is not human – it does not have emotional attachments, it does not have childhood memories, it does not have biochemical reactions. This does not mean, however, that we must shy away from providing software with the ability to make and justify subjective decisions. If the claims that it makes about its preferences are *consistent*, *defensible* and *reasonable*, we believe that this will add to the perception of the software being creative without deceiving the observer about the software's lack of humanity.

We describe here a system that can generate simple snippets of code, which we call *preference functions*, that take as input two objects of some type and express an ordering on them – in other words, they express what amounts to a *preference* between the two objects. This system works by evolving code segments, using a particular combination of metrics, which we also introduce here, as a fitness function. These metrics have been carefully designed to be domain agnostic, and to limit our influence as designers on the output the system ultimately produces in terms of the subjectivity it expresses. While this process is not perfect, we believe this work represents an encouraging first step towards software making meaningful subjective decisions. To illustrate this, we provide several examples of generated functions in different domains, including colour selection and videogame design, that highlight how this technique might be used in software. We then discuss what further work is needed to integrate this technique into the framing and context of computationally creative software.

Background

Framing and Subjectivity

Framing is the name given to the process by which software produces text or perhaps other content to provide context to a generated artefact. Thus far in Computational Creativity this generally takes the form of a 'wall text'-like commentary that appears alongside the artefact in order to help explain the creative process, as in (Colton, Goodwin, and Veale 2012). According to (Colton, Charnley, and Pease 2011), the authors claim that the act of framing can increase the 'value' of generative acts undertaken by software in several ways, one of which is 'by providing calculations about the concepts/expressions [in an artefact] with respect to the aesthetic measures'. In (Colton, Goodwin, and Veale 2012), for example, the software generates commentaries which explains why it chose particular poetic styles or focused on particular words.

In (Charnley, Pease, and Colton 2012), the authors consider three particular aspects of a creative work that framing can tackle: motivation, intention and process. The authors summarise these as 'Why did you do that?', 'What did you mean when you did that?' and 'How did you do that?' respectively. Of motivation, they say:

> *[it is] distinctly human in nature and it currently makes limited sense to speak of the life or attitudes of software in any real sense.*

However, the authors also point out later that 'framing need not be factually accurate', and that 'the motivation of a software creator may come from a bespoke process which has no basis in how humans are motivated'. We claim that it is reasonable for software to possess arbitrary or subjective preference about elements of its creative process, for the purposes of framing and justifying its motivation and output. The technique we outline in this paper has no basis in how people are motivated, as in the quote above, but it does aim to offer a form of motivation for software's actions that is satisfying to the observer and may withstand limited interrogation through framing or even dialogue.

Randomness and Believability

In (Colton and Wiggins 2012) the authors define Computational Creativity as the creation of systems which 'exhibit behaviours that unbiased observers would deem to be creative' (paraphrased). The mention of unbiased observers is crucial to the definition, since Computational Creativity is highly reliant on the *perception* of creativity. A common criticism of creative software is that the designer of the software is a major contributor to the software's creativity. (Colton 2009) proposes a process of 'climbing the meta-mountain' to overcome this, whereby creative software is iteratively improved to remove the influence of the original designer on the software, instead adding in new subsystems which take the place of the designer's involvement and allow the software to make the same decisions for itself.

The danger of removing designer influence for removal's sake is that the system that replaces the designer's involvement may not actually increase the perception of creativity. There is anecdotal evidence to suggest that people distrust the actions of software, even in cases where the software is proactively explaining that its decisions were intelligently motivated. The work described in (Cook and Colton 2014), for example, provoked an angry response from one member of the public who wrote 'AI, or just basic random number generation?' in response to the software framing its choice of a piece of music.

There are many explanations for why people might be biased against software in some instances, one being that they have good cause to be suspicious: random choice *is* used very often in the design of intelligent systems, including those in Computational Creativity. Moreover, as we have already stated, researchers are not afraid to have their software tell stories that are 'not factually accurate' in order to explain their decisions. This is not a dying practice, either: examining system description papers from the 2014 International Conference on Computational Creativity alone, we identified seven systems which explicitly mention random decision-making in their description (omitting cases where random selection might be part of a search-specific process, such as evolution) such as (Rashel and Manurung 2014), a poetry generator which randomly selects an output from any poems which meet a minimum quality, or (Tomašič, Žnidaršič, and Papa 2014) which breaks ties in slogan selection using random choice. Other systems described relying on hand-crafted metrics for making subjective decisions which inherit their decision-making capacity directly from the system's designer.

We believe that the underlying cause for this bias against software making decisions independently is not that people believe that software *cannot* make such decisions, but rather that random choice is not satisfying as a context for these decisions. Random choice cannot be interrogated or understood, does not form a long-term pattern of decision-making, and is also not something that people often do – even when people may in fact be making pseudorandom decisions, we often justify them post-hoc, particularly in the case of creative activity – see (Charnley, Pease, and Colton 2012) for examples. Most importantly, random choice cannot be easily framed through commentary on a creative artefact, because it has no context to reveal. This limits the software's ability to explain itself after the fact.

A System For Generating Preferences

If we acknowledge that inheriting decisions from a person damages the perception of software as being creative, but also accept that random decision-making is unsatisfying and can be equally damaging to perceptions, it leaves us in an awkward position whenever our software must tackle decisions which are subjective or where the factors involved are hard to quantify. Ideally, we would like our software to be able to provide meaningful reasons for small, subjective decisions. By meaningful, we mean that the decision is *defensible* in some way: there is a reasoning behind it, even if that reasoning is ultimately arguable (as subjective opinions often are, by their nature). In this section we will describe an evolutionary system that generates code to provide the basis for such decisions, with the primary aim being that these decisions are defensible, despite being subjective.

The system we describe here generates what we call *preference functions* – small snippets of code which express a preference of some kind between two objects of the same type. They are based on the concept of Comparators in Java which are used to express orderings over lists. A Comparator takes two objects and returns either -1, 0 or 1 if the first object is less than, equal to, or greater than the second object respectively according to some ordering. Our functions act similarly, where a preference can be thought of as an ordering over the set of objects of a particular type. More formally, we define a preference function p as a function which takes two arguments t_1, t_2 of type T, and returns one of three integer values $r \in \{-1, 0, 1\}$. The return value indicates the following three situations:

$$p(t_1, t_2) = \begin{cases} 1 & t_1 >_p t_2 \\ 0 & t_1 =_p t_2 \\ -1 & t_1 <_p t_2 \end{cases}$$

Where $*_p$ is an ordering according to preference, i.e. $t_1 <_p t_2$ states that t_1 is preferred over t_2 in some way. The comparator can therefore be used to order a list L_T of objects of type T.

Before we describe the operation of the evolutionary system, we will go into some detail about the fitness function that evaluates a particular preference function. Earlier, we claimed that our intention was to limit the influence of a person's opinion over the system's eventual decisions. Below we will propose metrics which direct the search for preference functions – in some sense we are defining the kinds of preferences the system looks for. We will try to justify our decisions and show that these metrics are flexible, domain-agnostic, and aim for defensibility without specifying anything about what kinds of preference should be expressed.

Fitness Metrics

In this section we describe several metrics that can be used to assess certain qualities of a preference function. This does not make a judgement about the 'goodness' of the preference expressed; a preference function which scores higher on these criteria is not objectively better than one with lower scores. Rather, we aim to identify meta-level properties of preference functions in order to search for spaces of interesting, valid or defensible preferences. As a result, we've tried to avoid the use of emotive or judgemental vocabulary when describing the metrics.

Specificity The *specificity* of a preference function p for a set of objects O is defined as:

$$1 - \frac{|N_p|}{|P|}$$

with

$$\begin{aligned} P = &\quad \{(a,b) \mid (a,b) \in O \times O \wedge a \neq b\} \\ N_p = &\quad \{(a,b) \mid (a,b) \in P \wedge p(a,b) = 0\} \end{aligned}$$

In other words: the specificity of a preference function for a particular list of objects is the proportion of the list for which it returns a nonzero result, i.e. a definite preference. Note that this excludes identity preferences (you can't prefer something to the same thing), but it does not assume transitivity on p and it includes reflexive preference, i.e. $p(a,b)$ and $p(b,a)$.

Transitive Consistency The *transitive consistency* of a preference function p for a set of objects O is defined as:

$$\frac{|T_p|}{|Q|}$$

with

$$\begin{aligned} Q = &\quad \left\{ (a,b,c) \;\middle|\; \begin{array}{l} (a,b,c) \in (O \times O \times O) \\ \wedge\, a \neq b \wedge b \neq c \wedge a \neq c \end{array} \right\} \\ T_p = &\quad \{(a,b,c) \mid (a,b,c) \in Q \wedge tight_p(a,b,c)\} \end{aligned}$$

Where $tight_p$ holds for a triple (a,b,c) if the triple is transitively non-contradictory under the preference function p. That is:

$$a \geq_p b \wedge b \geq_p c \implies a \geq_p c$$

In other words, transitive consistency is a measure of how much the decisions made by the preference function confirm one another when compared alongside each other. A high transitive consistency means that the preference is well-ordered. As with other metrics, this is not inherently good or bad. Low transitive consistency can imply that the preference selects based on unconnected or competing features in the artefacts, which is not uncommon in everyday preferences.

Reflexivity The *reflexivity* of a preference function p for a set of objects O is defined as:

$$\frac{|P_r|}{|P|}$$

with

$$P_r = \quad \{(a,b) \mid (a,b) \in P \wedge p(a,b) = -p(b,a)\}$$

In other words, reflexivity is a measure of how dependent p is on the ordering of its arguments. A high reflexivity suggests that the preference being expressed is not dependent on the arguments being supplied to it. This metric is useful specifically because of how we generate code, since it is possible to generate functions which make decisions based on the ordering of their parameters (always preferring the first parameter, for example). High reflexivity means that no parameter is preferred over another simply because of the order they are passed in.

Agreement Agreement is a special metric for comparing two preference functions. This isn't used to generate preferences, but can be used to compare them, or generate functions in opposition to one another. Two preference functions p_1 and p_2 are said to be in {k,n}-agreement iff:

$$k \leq \frac{|P_{p_1,p_2}|}{|P|}$$

with

$$P_{p_1,p_2} = \quad \left\{ (a,b) \;\middle|\; \begin{array}{l} (a,b) \in P \wedge \\ \begin{pmatrix} p_1(a,b) = p_2(a,b) \vee \\ p_1(a,b) = 0 \vee p_2(a,b) = 0 \end{pmatrix} \end{array} \right\}$$

With P and O as before, and where $|O| = n$. In other words, agreement measures how closely the definite decisions of two preference functions are the same – the proportion of pairs (a, b) for which p_1 and p_2 either evaluate the same value or one of them is zero, is greater than k for a list of objects of size n. This is a good measure of how close two preference functions are on the same sample of inputs.

To summarise, the first three metrics judge preference functions along several dimensions: how often they make a definite (nonzero) judgement on two objects, how consistent their judgements are across a list of objects, and how dependent the decisions are on the random ordering of inputs. The final metric, agreement, can be used to model how similar or dissimilar two preference functions are on the same set of inputs. We will now describe the preference function generation system, which uses a combination of the first three metrics in its evaluation.

Representing Preference Functions

In the following subsections we describe an evolutionary system implemented in C# which uses the CodeDOM API to generate code segments which act as the body of a preference function. The language and library are arbitrary choices for convenience, and should be transferable to any platform for which code generation is possible.

CodeDOM is an API within Microsoft's .NET library that allows for the high-level (and extremely verbose) representation of code, which can later be exported to `.cs` files and then compiled into executable assemblies within C#. The code below is equivalent to `(p && q)` in C#, where `p` and `q` are local variables:

```
new CodeBinaryOperatorExpression(
  new CodeVariableDeclarationStatement("p"),
  CodeBinaryOperatorType.BooleanAnd,
  new CodeVariableDeclarationStatement("q"));
```

CodeDOM compilation units can be exported to code programatically, and these files can be compiled and executed at runtime using C# CodeProvider classes. CodeDOM represents almost every aspect of the C# language, but for our purposes we do not extend the code generation to the entire C# specification, as this yields diminishing returns and is too large a state space for this stage of experimentation. Generation of the preference functions described here is limited to:

Expressions

- Primitive expressions containing `int`, `bool` or `String` types.
- References to the object parameters given to the function.
- Boolean binary operations including `&&` and `||`.
- Numeric binary operations.
- Type casts between certain compatible types.
- Array index references.
- Field accesses in objects.

Statements

- Conditional control flow statements (`if` statements)
- Assignment
- Return of either -1, 0 or 1.

We define a *code segment* as a list of one or more statements. This can be put inside a CodeDOM representation of a method. We define a generic abstract template class which defines a method `public int compare(int a, int b)`[1]. Generated code segments are put inside a class which extends this abstract template, providing an implementation for the `compare` method.

Evolving Preference Functions

A population of code segments is randomly generated, and evaluated using the following objective function:

$$\begin{aligned} fitness(p) &= 0.5 \times reflexivity(p) \\ &+ 0.25 \times specificity(p) \\ &+ 0.25 \times consistency(p) \end{aligned}$$

This objective function was developed through manual experimentation, but again we stress that this is not considered optimal in any way. Specificity may be more important in some domains, while totally unimportant in others, for instance – it depends on the nature of the preference functions that the programmer wishes to generate. In our case, reflexivity was found to be important in ensuring a perception of defensibility in the resulting preference functions. A high weighting for reflexivity might be preferable in many application domains, we will determine this in further development and use of these criteria, and we expect variation to be found in the other metrics as well according to the needs of the individual system.

Because of the nature of code generation, particularly our code generator's implementation, it is possible for a code segment to either fail compilation, or to throw exceptions during evaluation. We catch and ignore any errors in this process and assign a negative fitness to the code segment.

Crossover of two code segments uses one-point crossover on the list of code statements making up the segment. This is currently acceptable for the subspace of the C# specification we cover, although once local variables are introduced, this approach will need revision to avoid constantly introducing scope errors (where a local variable is referenced in the latter half of a function but its declaration was not carried over during crossover). Mutation is applied by randomly regenerating one of the code statements in the list of statements. As with crossover, once the system's focus moves to more complex method constructions, a finer-grained mutation process may be required that is capable of making small changes to individual statements in a method.

In order to speed up the evolutionary system, we compile an entire population of preference functions simultaneously, passing each comparator as a separate file along with the template comparators they inherit from. If errors are thrown during the compilation of a particular comparator, they do not affect the compilation of the other files passed. Testing

[1]The types of the parameters to the function are changed from `int` depending on what type the system is evolving comparators for.

of this method showed it was far more efficient than single-file compilation, even when done in parallel, because most of the overhead of compilation is in initialising and shutting down the compiler itself. This may change in the case of generating extremely large code blocks, but we do not expect it to be an issue in the near future.

Results

In this section we give several example results from the system, for different domains. We begin with some simple examples for comparing integers, then show two more applied examples: preference functions which decide between colours expressed in RGB format, and preference functions which compare pieces of game content embedded as part of a simple videogame. In each case, we give the code of the preference function, and an English description of the code.

These are hand-selected preference functions, however the curation coefficient – that is, the proportion of the system's output which we would be happy to show to others, as in (Colton and Wiggins 2012) – is extremely high. So far we have not seen any high-fitness comparators (>0.95 fitness) that would not act as justifiable, if simple, preferences in some way. Curation is necessary only to avoid showing the same function twice, because the system frequently generates comparators with identical functionality but very different code, as we do not yet implement a novelty search (Lehman and Stanley 2010).

Basic Preference Examples

The results shown in Figures 1 and 2 were generated with a population of 20 code segments, a test set of 100 random integers in the range $\{-500, 500\}$ to evaluate the preference functions, and 15 generations of evolution. We found this to be sufficient to evolve high-fitness (0.95 or higher) functions that compared integers.

Figure 1 shows a preference function which prefers negative numbers over positive ones. This is expressed in a rather awkward way: by adding the two arguments together and comparing them with one of the arguments on its own. The else case in this conditional statement returns the opposite ordering instruction (`-1`), meaning that this function has a high consistency while also being precise.

Figure 2 shows a more standard ordering on integers, from smallest to largest. Both this method and Figure 1 have large amounts of unreachable or redundant code. This is expected, given that the system is concerned with the function of code rather than its design. The unnecessary code is not impossible to filter out with the right interpretation of compiler messages, since the C# compiler recognises many of these issues and will present warnings to the system when attempting to compile. We touch on this topic in the discussion section.

In a further experiment, we expanded the expressivity of the code generation to include the `char` primitive type as well as the notion of casting to a type. Evolving high-fitness results for this target domain was more difficult and required a larger evolutionary run than with integer types. We ran populations of 30 code segments, a test set of 100 random `chars` whose ASCII codes fall in the range $\{0, 128\}$ to evaluate the preference functions, and 15 generations of evolution. Figure 3 shows a function which sorts `chars` in reverse lexicographic order. We increased the population size because usable preferences were proving difficult to evolve – as one can see, this is most likely because type casting was required to produce the simplest preference functions, which makes the code much longer and therefore harder to evolve.

```
public int compare(int i, int j) {
    if ((i < i)) {
        return 0;
        return 0;
    }
    if (((j + i) < j)) {
        i = i;
        return -1;
    }
    else {
        j = -491;
        return 1;
    }
    return 0;
}
```

Figure 1: If `i` is negative, it is preferred over `j`; the second conditional check is true if $i < 0$.

```
public int compare(int i, int j) {
    if ((i <= j)) {
        return -1;
        j = ((i * 335) % j);
    }
    else {
        return 1;
        j = j;
    }
    return 1;
    return -1;
}
```

Figure 2: Orders numbers from largest to smallest. The first conditional returns a reverse ordering (-1) if the first argument is smaller than the second. Note the copious amount of unreachable code. This constitutes a compile-time warning in C#, which is suppressed here.

Object Preferences

Figure 5 shows a preference function evolved for comparing a more complex type – in this case, an object with four fields representing a Monster from a simple game. The class skeleton for the object is shown in Figure 4. These examples were also evolved with a population of size 40, run for 30 generations, with a test set size of 100. Evolving preference functions for objects gives the system a wider state space to explore with more interesting comparisons available to it, with the potential to generate preference functions which compare along two axes simultaneously.

We are building a prototype game, *I Like This Monster* that uses preference generation as part of a process of automated game design. Choosing one particular kind of game

```
public int compare(char i, char j) {
    if ((((int)(j)) <= ((int)(i)))) {
        return 1;
        return -1;
    }
    else {
        i = ((char)(((((int)(i)) -
        ((int)(i))) * (((int)(j)) -
        ((int)(j))))));
        return -1;
    }
    return 0;
    return 0;
}
```

Figure 3: Reverse lexicographic ordering on characters. Note that explicit casts to `int` types has caused a lot of excess bracketing.

element over another has a large component of subjectivity to it, particularly if the game content is already balanced for difficulty and fun. Rather than randomly choosing certain game elements, or choosing them according to a fixed designer preference, the game generates a preference for certain game elements like monsters. This preference is then used to select from a database of pre-generated game content to decide what is included in the game. This is analogous to generating multiple poems and choosing between them as in (Rashel and Manurung 2014) – but unlike random choice, the use of preference functions means that the decision can be *framed* and given a justification. We discuss how one might generate text from preference functions below.

For a more visual example of a preference function, Figure 8 shows another example. In this case, a preference function is generated for an object representing RGB colours, with three `int` fields representing each colour component. A preference function was generated which prefers colours with more red in them. Figure 8 shows the effect this has: the top row shows a randomly generated line of RGB colours, and the bottom row shows the same line ordered from least preferred on the left to most preferred on the right. The preference is very simplistic – it doesn't quite correlate to a visual language of 'redness', but the software can justify its decision on a code level even if it does not directly corresponding to visual processing in people.

In all of the results given in this section, we found that reflexivity is the metric which was maximised quickest. This is likely because it is the simplest to satisfy, as it primarily safeguards against particular bad patterns of code being generated (as long as the function does not return an answer based on the ordering of the arguments, it is always maximised). If the target is high specificity, this is often maximised next, as this requires the preference function simply return nonzero values. However, more complex specificity requirements may require branching and non-constant return statements. In this case, it is much harder to maximise than transitive consistency. These observations largely apply here because the domains we are considering are relatively simple and the preference functions we are generating are low in complexity and length. We expect this to change in future – functions which compare multiple variables simultaneously, for example, are far more likely to be transitively inconsistent, while functions which return variable values or have high branching are more likely to have lower specificity. This raises the question of how to find these functions over evolving simpler preferences – it may be that additional 'interestingness' metrics are required, or it may simply be that asking for longer preferences or a novelty search powered by agreement will be enough to promote the evolution or more complex preferences.

```
public class Monster{
    public string name;
    public int health;
    public int damage;
    public boolean poisonous;
}
```

Figure 4: A dummy class specification used for generating preference functions. `health` cannot have a negative value, but `damage` can (some monsters heal by attacking).

```
public int compare(Monster i, Monster j){
  i.name = j.name;
  if ((j.health > i.health)){
    i = j;
    return 1;
  }
  else{
    return -1;
    j.name = j.name;
  }
  i.damage = (i.health / i.health);
}
```

Figure 5: An ordering on Monster objects based on their `health` variable.

Related Work

No work we are aware of directly tackles the problem of generating meaningful, defensible preferences for creative agents across arbitrary domains. However, the idea of

```
public override int compare(RGB i, RGB j){
  if(((j.r * j.r) > (j.r + (i.r - j.r)))){
    j = i;
    return -1;
  }
  else {
     return 1;
  }
}
```

Figure 6: An ordering on RGB objects based on their `r` (red component) variable.

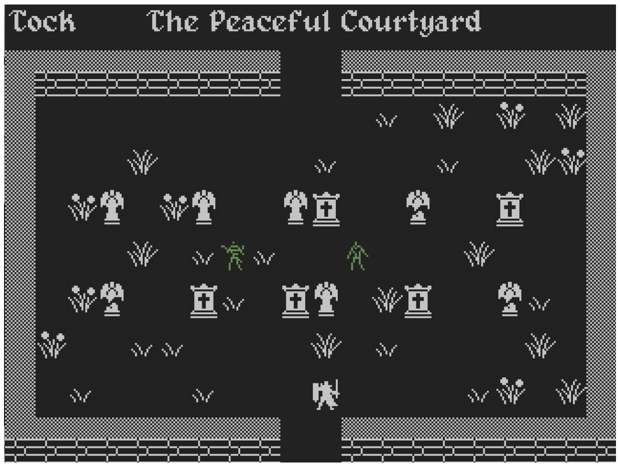

Figure 7: A screenshot from *I Like This Monster*, showing a level where a particular kind of enemy - poisonous creatures - has been selected because of a preference for monsters with the `poisonous` field set to true.

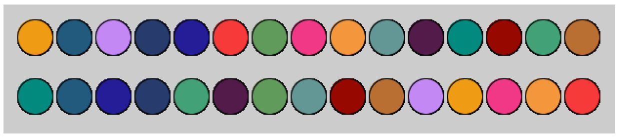

Figure 8: A random colour palette (top row) and an ordering of the same palette according to a preference about RGB colors which prefers colours with more red in them (bottom row, preferred colours towards the right).

computationally representing subjective decisions has some precedent. In (Saunders and Gero 2001) the authors describe a community of creative agents which are designed to have some concept of novelty and interestingness. Each agent possesses a neural network which learns by viewing artworks generated by agents in the community. This can be used to gauge novelty for a given artwork by assessing how much the artwork differentiates itself from the network's current state. Interestingness is based on a Wundt curve calculation in which the most interesting artefacts lie between the extremes of high and low novelty.

(Saunders and Gero 2001) can be seen as a form of preference modelling, in that the agents are armed with a way of making decisions about creative works, if we interpret interestingness to be a subjective quota. Our work is different in a few important ways: it generates a range of preferences based on different factors that vary according to the objects being considered, whereas the community of agents only work in the realm of novelty. The work is also more prescriptive, in our opinion, than the metrics we propose - although we should stress that the authors do not claim to be investigating the generation of varied preferences, the work has other objectives, but we have cited it here as an interesting piece of related work.

Similarly, (Maher, Fisher, and Brady 2013) presents a computational model of surprise, which could be considered to be a form of preference if applied to selection or evaluation (as the authors propose). Similar to (Saunders and Gero 2001), we differentiate ourselves from this work primarily because our aim is to produce a higher-level system which can generate a variety of preferences based on different factors, rather than primarily basing it on surprise or novelty.

The automatic creation of code by software is not a new concept. Code generation, or 'unrolling' of code, is a common concept in software engineering, used for purposes such as optimisation, or the automatic reconfiguration of code in response to dynamically changing execution environments. This is often highly template-based, and the code is generated for precise functional objectives that are normally known well in advance.

Code-generating systems also exist in artificial intelligence. Machine learning software, for example, can be viewed as producing programs as their primary output. Decision trees, neural networks or inductive logic programs can all be seen as forms of computer programs, sometimes (such as the case of ILP or evolutionary programming) quite explicitly. Machine learning techniques have been seen in Computational Creativity on many occasions. For instance, (Morris et al. 2012) uses machine learning as the basis for a computationally creative soup recipe inventor, trained on a corpus of existing soup recipes, and (Colton 2008) uses machine learning in a module within The Painting Fool, a computationally creative artist.

The generation of code is perhaps most explicitly present in Computational Creativity in (Cook et al. 2013), in which we presented Mechanic Miner, a system which explores, modifies and executes the codebase for a simple videogame, in order to discover new concepts for game mechanics and rules. The system was capable of generating single lines of code, modifying the existing game's code to include this new instruction, and then playing the game to evaluate the effect of the generated code on gameplay. In doing this, the system rediscovered several existing game design concepts, previously invented and used by game designers. It was also capable of surprising us as the creators of the system, by presenting solutions which were highly unexpected or took advantage of the system's detailed use of code to perform unexpected operations on the target videogame. This notion of generating directly executable, readable program code in an everyday programming language is one of the motivations for the work we have described here.

Discussion

The preference functions presented in this paper represent a first step towards a system which can reliably generate interesting preferences for arbitrary targets. We believe it represents a promising new avenue for exploration, and one that could greatly enhance the quality of framing that Computational Creativity systems are able to provide.

Generating code which claims to represent 'preference' is potentially controversial. The reason for many decisions being randomised or guided by hand-designed heuristics in the first place is that software does not hold personal opinions and is not human. We would argue, however, that we are in the business of perception – recall the definition of Computational Creativity from earlier as being dependent on 'unbiased observers'. Whether we like it or not, our software is judged on how it presents itself, and our first-hand experience of building systems and presenting them to the public has shown us that random decision-making and heuristics inherited from people are as damaging to expectation and perception as any amount of personification.

Furthermore, we would argue that having software express a preference is not necessarily in bad faith. Representing a random decision as having a basis in personhood is deceiving the observer, but with a preference function there is a chain of reasoning, a process that is itself accountable and can be framed, that shows where this preference has originated from. This preference can be interrogated, an observer can present new examples to it to try and better understand what it prefers and why. The software is not claiming to have an emotional basis for this – it is simply stating a preference that it used to guide its decision-making process. Of course this does not offer a perfect solution to all the problems of subjective decision-making in creative software, but we believe it offers a new way of exploring the issue.

It is worth noting that these preferences are, in some ways, equivalent to random choice. They are arbitrary, domain-agnostic, they do not care about their impact on the viewer (unless we used the agreement to be contrarian, perhaps). We are not claiming here that these preferences provide a benefit to the code over randomly choosing something, nor do we even claim that it makes the system more creative in terms of its functionality. We do believe, however, that they provide a benefit to the *perception* of the software as creative if its decisions can be justified, if we can claim that no random number generation is involved, and if its decision-making process can be inspected and interrogated by observers.

There are several important areas of future work to be undertaken in order for the system described in this paper to be able to work in large computationally creative systems. Some of these topics have already appeared earlier in this paper. Firstly, the system should be expanded with a larger state space to explore in terms of code generation, so that more complex functions can be generated. This may be possible with existing techniques simply by applying it at a larger scale, however the state space explosion is significant once more complex programming features – like method invocation – are taken into account. It may be that evolution is not the best approach for generating code at this scale, or that the process requires alteration in order to be more efficient for this kind of optimisation problem.

A second point of future work is automatic simplification of generated preference functions. This is an achievable goal, and many optimisation processes for program compilation already do this. We mention it here because it is particularly important for code generation in the context of Computational Creativity, as we explained in (Cook et al. 2013). Compressing a piece of code by removing unreachable or non-functional code makes it easier to understand, easier to compare, and also has the important side effect of making it easier to explain, which is a third future work task.

Being able to explain the function of a piece of code is crucial to this work – in the examples we gave in the Results section there was a lack of textual framing to the visual examples. In some senses it is possible to interpret the effect of the preferences simply by looking at the content produced, but in general it is desirable to be able to have the system itself express 'I prefer redder colors'. Producing English renderings of the function of code is complex – we are currently exploring possibilities which use some metadata tagging on the code prior to generating preferences, but there are many more and better approaches yet to be discovered.

Finally, representing preference functions in a higher-level mathematical language may be advantageous for this work. Many of the problems we have encountered are direct consequences of the code-based representation, such as the presence of unreachable code and functionally identical generation. We hope to look into more abstract representation formats for future versions of our system.

Conclusions

In this paper we introduced a series of criteria for assessing functions that describe preferences, motivated by a desire to provide non-random justifications for small creative decisions that don't rely on other people. We showed how an evolutionary system can use these criteria as the basis for a fitness function that evolves code which act as preference functions. We gave examples of preference functions we evolved using these criteria for comparing various types, including videogame content and colours, and discussed the issues it raises for Computational Creativity, in terms of the code itself and the nature of generated preferences.

The perception of creativity in software is a defining problem for our field. We hope that the work we have described here offers a new avenue to explore for framing decisions made by the software we build. Even the smallest of decisions are affected by people's perceptions of software as arbitrarily random, or clones of their designers. We believe that the future of decision-making in software lies beyond random choice and modelling human opinion – we need to give our software independence and remove the influence of other people on it. We acknowledge that we have by no means managed to remove ourselves from the process of decision-making – we have designed the system which produces preference functions, defined its metrics and provided it a fitness function. But we hope that we have offered a way to take one step further into the background, leaving our software to stand alone at the fore.

Acknowledgments

The authors would like to thank Adam Smith and Rob Saunders for discussions about code generation, and Alison Pease for input on an earlier version of this paper. Thanks to the reviewers for very thorough reviews which helped improve this paper. This paper is a difficult balance of position paper and systems description, which made it hard to meet all of your comments, but we have taken them on board for our future work and writing, and we appreciate them greatly. This work was funded by EPSRC grant EP/L00206X.

References

Charnley, J.; Pease, A.; and Colton, S. 2012. On the notion of framing in computational creativity. In *Proceedings of the Third International Conference on Computational Creativity*, 77–81.

Colton, S., and Wiggins, G. A. 2012. Computational creativity: The final frontier? In *ECAI 2012 - 20th European Conference on Artificial Intelligence.*, 21–26.

Colton, S.; Charnley, J.; and Pease, A. 2011. Computational creativity theory: The FACE and IDEA descriptive models. In *Proceedings of the Second International Conference on Computational Creativity*, 90–95.

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

Colton, S. 2008. Experiments in constraint-based automated scene generation. In *Proceedings of the International Joint Workshop on Computational Creativity 2008*.

Colton, S. 2009. Seven catchy phrases for computational creativity research. In *Computational Creativity : An Interdisciplinary Approach*, Schloss Dagstuhl Seminar Series.

Cook, M., and Colton, S. 2014. Ludus ex machina: Building a 3D game designer that competes alongside humans. In *Proceedings of the Fifth International Conference on Computational Creativity*.

Cook, M.; Colton, S.; Raad, A.; and Gow, J. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *Proceedings of 16th European Conference on the Applications of Evolutionary Computation*.

Eigenfeldt, A.; Burnett, A.; and Pasquier, P. 2012. Evaluating musical metacreation in a live performance context. In *Proceedings of the Third International Conference on Computational Creativity*, 140–144.

Lehman, J., and Stanley, K. O. 2010. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation* 19(2):189–223.

Maher, M. L.; Fisher, D.; and Brady, K. 2013. Computational models of surprise as a mechanism for evaluating creative design,. In *Proceedings of the Fourth International Conference on Computational Creativity*, 147–151.

Morris, R. G.; Burton, S. H.; Bodily, P. M.; and Ventura, D. 2012. Soup over bean of pure joy: Culinary ruminations of an articial chef. In *Proceedings of the Third International Conference on Computational Creativity*.

Rashel, F., and Manurung, R. 2014. Pemuisi: a constraint satisfaction-based generator of topical indonesian poetry. In *Proceedings of the Fifth International Conference on Computational Creativity*.

Saunders, R., and Gero, J. S. 2001. The digital clockwork muse: A computational model of aesthetic evolution. In *The AISB'01 Symposium on AI and Creativity in Arts and Science, SSAISB*, 12–21.

Tomašič, P.; Žnidaršič, M.; and Papa, G. 2014. Implementation of a slogan generator. In *Proceedings of the Fifth International Conference on Computational Creativity*, 340 – 343.

Veale, T. 2013. Once more, with feeling! using creative affective metaphors to express information needs. In *Proceedings of the Fourth International Conference on Computational Creativity*.