Linear Genome Representation

Canonical Grammatical Evolution uses linear genomes (also called chromosomes) to encode genetic information [[[O'Neill & Ryan, 2003|References]]]. These linear genomes are then mapped via the use of a formal BNF-style grammar to produce a phenotypic output. All individuals in PonyGE2 have an associated linear genome which can be used to exactly reproduce that individual.

PonyGE2 contains a number of operators that manage linear genomes. These are discussed in later sections.

***NOTE** that in general the use of a linear genome does not allow for [[context-aware (or "intelligent") operations|Representation#context-aware-intelligent-operations]]. Although intelligent linear genome operators exist, e.g.* [[[Byrne et al., 2009|References]]], *they are not implemented here as similar functions can be performed in a more intuitive manner using derivation-tree based operations.*

Codon Size

Each codon in a genome is an integer value that maps to a specific production choice when passed through the grammar. When generating a codon to represent a production choice, a random integer value is chosen that represents that correct production choice. The maximum value a codon can take is set by default at 10000. This value can be changed with the argument:

```
--codon_size [INT]
```

or by setting the parameter `CODON_SIZE` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the maximum value a codon can take.

Genotype-Phenotype Mapping Process

The genotype is used to map the start symbol as defined in the grammar onto terminals by reading codons to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

```
Rule = c % r
```

where `c` is the codon integer value, `r` is the number of rule choices for the current non-terminal symbol, and `%` is the modulus rule (also called the mod rule).

Consider the following rule from a hypothetical grammar. Given the non-terminal `<op>`, which describes the set of mathematical operators that can be used elsewhere in the grammar, there are four production rules to select from. As can be seen, the choices are effectively labelled with integers counting from zero.

```
    <op> ::= +    (0)
           | -    (1)
           | *    (2)
           | /    (3)
```

If we assume the codon being read produces the integer 6, then:

```
6 % 4 = 2
```

would select rule (2) `*`. Therefore, the non-terminal `<op>` is replaced with the terminal `*` in the derivation string. Each time a production rule has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

Many-to-one Mapping

Since the genotype-to-phenotype mapping process in GE uses the modulus rule to map a gene to a production choice through a given production rule, it is possible for multiple genes/codons to select the same production choice from the same production rule. For example, consider the following production rule with three production choices:

```
<var> ::= x | y | z
```

For this production rule, *any* codon `c` that has a remainder of 0 (i.e. `c % 3 = 0`) will result in the first choice (`x`) being returned. Similarly, any codon with a remainder of 1 will return `y`, and any codon with a remainder of 2 will return `z`. This is called a "many-to-one" mapping, and is a key cornerstone of the flexibility of the genetic mapping process in GE [[[O'Neill & Ryan, 2003|References]]]. Many-to-one mapping allows for neutral mutations, degeneracy, and introns, among other things [[[O'Neill & Ryan, 2003|References]]].

Wrapping

During the genotype-to-phenotype mapping process, it is possible for the genome to run out of codons before the mapping process has terminated. In this case, a *wrapping* operator can applied which results in the mapping process re-reading the genome again from the start (i.e. wrapping past the end of the genome back to the beginning). As such, codons are reused when wrapping occurs. This is quite an unusual approach in evolutionary algorithms as it is entirely possible for certain codons to be used two or more times depending on the number of wraps specified. This technique of wrapping the individual draws inspiration from the gene-overlapping phenomenon that has been observed in many organisms [[[Lewin, 2000|References]]]. GE works with or without wrapping, and wrapping has been shown to be useful on some problems [[[O'Neill & Ryan, 2003|References]]], however, it does come at the cost of introducing functional dependencies between codons that would not otherwise arise.

By default, wrapping in PonyGE2 is not used (i.e. the `MAX_WRAPS` parameter is set to 0). This can be changed with the argument:

```
--max_wraps [INT]
```

or by setting the parameter `MAX_WRAPS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the desired maximum number of times the mapping process is permitted to wrap past the end of the genome back to the beginning again.

***NOTE** that **permitting** the mapping process to wrap on genomes does not necessarily mean it **will** wrap across genomes. The provision is merely allowed.*

Invalid Individuals

In GE each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is being applied, it may result in the selection of a different production rule. This feature is referred to as *intrinsic polymorphism*. What is crucial however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is the case because the same choices are made each time. It is possible that an incomplete mapping could occur, even after several wrapping events, and typically in this case the mapping process is aborted and the individual in question is given the lowest possible fitness value. The selection and replacement mechanisms then operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules repeatedly. For example, consider an individual whose full genome consists of three codons `[3, 21, 9]`, all three of which map to production choice 0 from the following rule:

```
<e> ::= (<e><op><e>) (0)
      | <e>          (1)
      | <op>         (2)
```

Even after wrapping, the mapping process would be incomplete and would carry on indefinitely unless terminated. This occurs because the non-terminal `<e>` is being mapped recursively by production rule 0, i.e., `<e>` becomes `(<e><op><e>)`. Therefore, the leftmost `<e>` after each application of a production would itself be mapped to a `(<e><op><e>)`, resulting in an expression continually growing as follows:

```
((<e><op><e>)<op><e>)
```

followed by

```
(((<e><op><e>)<op><e>)<op><e>)
```

and so on.

Since the genome has been completely traversed (even after wrapping), and the derivation string (i.e. the derived expression) still contains non-terminals, such an individual is dubbed *invalid* as it will never undergo a complete mapping to a set of terminals. Invalid individuals are given a `NaN` fitness in PonyGE2, and are counted in the stats. All population statistics in PonyGE2 are compiled ignoring these `NaN` fitness individuals.

Reducing Invalids

An upper limit on the number of wrapping events that can occur is imposed (as specified by the parameter `MAX_WRAPS`, detailed above), otherwise mapping could continue indefinitely in the invalid case. During the mapping process therefore, beginning from the left hand side of the genome codon integer values are generated and used to select rules from the BNF grammar, until one of the following situations arise:

1. A complete program is generated. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar.
2. The end of the genome is reached, in which case the wrapping operator is invoked. This results in the return of the genome reading frame to the left hand side of the genome once again. The reading of codons will then continue, unless an upper threshold representing the maximum number of wrapping events has occurred during this individual's mapping process.
3. In the event that a threshold on the number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual is assigned a `NaN` fitness value.

To reduce the number of invalid individuals being passed from generation to generation various strategies can be employed. Strong selection pressure could be applied, for example, through a steady state replacement. One consequence of the use of a steady state method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals. Alternatively, a repair strategy can be adopted, which ensures that every individual results in a valid program. For example, in the case that there are non-terminals remaining after using all the genetic material of an individual (with or without the use of wrapping) default rules for each non-terminal can be pre-specified that are used to complete the mapping in a deterministic fashion. Another strategy is to remove the recursive production rules that cause an individual's phenotype to grow, and then to reuse the genotype to select from the remaining non-recursive rules. Finally, the use of genetic operators which manipulate the derivation tree rather than the linear genome can be used to ensure the generation of completely mapped phenotype strings.

Derivation Tree Representation

During the [[genotype-to-phenotype mapping process|Representation#genotype-phenotype-mapping-process]], a derivation tree is implicitly generated. Since each production choice in standard GE requires a codon (note that [[unit productions consume codons in PonyGE2|Grammars#a-note-on-unit-productions]]), each chosen production choice can be viewed as a node in an overall derivation tree. The parent rule/non-terminal node that generated that choice is viewed as the parent node, and any production choices resultant from non-terminals in the current production choice are viewed as child nodes. The depth of a particular node is defined as how many parents exist in the tree directly above it, with the root node of the entire tree (the start symbol of the grammar) being at depth 1. Finally, the root of each individual node in the derivation tree is the non-terminal production rule that generated the node choice itself.

While linear genome mapping means that each individual codon specifies the production choice to be selected from the given production rule, it is possible to do the opposite. Deriving an individual solution purely using the derivation tree (i.e. *not* using the [[genotype-to-phenotype mapping process|Representation#genotype-phenotype-mapping-process]]) is entirely possible, and indeed provides a lot more flexibility towards the generation of individuals than a linear mapping.

In a derivation tree based mapping process, each individual begins with the start rule of the grammar (as with the linear mapping). However, instead of a codon from the genome defining the production to be chosen from the given rule, a random production is chosen. Once a production is chosen, it is then possible to retroactively *create* a codon that would result in that same production being chosen if a linear mapping were to be used. In order to generate a viable codon, first the index of the chosen production is taken from the overall list of production choices for that rule. Then, a random integer from within the range `[number of choices : number of choices : CODON_SIZE]` (i.e. a number from `number of choices` to `CODON_SIZE` with a step size of `number of choices`). Finally, the index of the chosen production is added to this random integer. This results in a codon which will re-produce the production choice. For example, consider the following rule:

```
<e> ::= a | b | c
```

Now, let us randomly select the production choice `b`. The index of production choice `b` is 1. Next, we randomly select an integer from within the range `[3, CODON_SIZE]`, giving us a random number of 768. Finally, we add the index of production choice `b`, to give a codon of 769. In this manner it is possible to build a derivation tree, where each node will have an associated codon. Simply combining all codons into a list gives the full genome for the individual.

Importantly, since the genome does not define the mapping process, it is not possible for "[[invalid|Representation#invalid-individuals]]" solutions to be generated by derivation tree based methods.

Context-Aware "Intelligent" Operations

Since production choices are not set with the use of a derivation tree representation (i.e. the production choice defines the codon, rather than the codon defining the production choice), it is possible to build derivation trees in an "intelligent" manner by restricting certain production choices and using context-aware operators. For example, it is possible to force derivation trees to a certain depth by only allowing recursive production choices to be made until the tree is deep enough that branches can be terminated at the desired depth. This is the basis of "intelligent" derivation methods such as [[Ramped Half-Half (or Sensible) initialisation|Initialisation#ramped-half-half]].

It is also possible to perform intelligent variation operations using derivation tree methods. For example, subtree [[crossover|Variation#subtree]] and [[mutation|Variation#subtree-1]] can be controlled by only selecting desired types of sub-trees for variation. Such operators are included in PonyGE2, and are described in the [[Variation]] section.