Grammatical Evolution

Rui Mendes

Grammatical Evolution

- Instead of using trees, genomes map into a grammar that defines the program
- Uses BNF Grammars consisting of the tuple $< T, N, P, S >$ where

  T : is the set of Terminals

  N : is the set of Non-Terminal

  P : is the set of Production Rules

  S : is the start symbol ($S \in N$)

Advantages

- Can evolve programs in any language or complexity
- Ability to evolve solutions for any problem where the solution can be encoded by a grammar.
- Separation of search (genome manipulation) from solution representation (grammar mapping).
- Support for evolving programs in arbitrary languages.

BNF Example for Symbolic Regression

```
<expr> ::= <expr> <op> <expr>
     |  (<expr> <op> <expr>)
     |  <fun> (<expr>)
     |  <var>
<op>   ::= + | - | * | /
<fun>  ::= Sin | Cos | Tan
<var>  ::= X
```

Characteristics

- Genotype to phenotype mapping
- Genotype is usually a sequence of integers
- Phenotype is a valid phrase in a given grammar
- This may be a program
- Genome indicates how the program is built using the BNF grammar
- Anything that can be specified with a gramar can be evolved, like:
  - programs
  - neural networks
  - graphs
- Can handle different types (e.g., Boolean, Integer, String)

Genotype to phenotype encoding

- Several genes map to the same phenotype
- Genome is usually a sequence of integers
- Each integer indicates which production to choose

Example

- Given the individual:

  280 45 127 29 59 163

- The NT <expr> has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
     |   (<expr> <op> <expr>)
     |   <fun> (<expr>)
     |   <var>
```

- Using the first codon 280, we get $280 \mod 4 = 0$
- Thus, we will use <expr> <op> <expr>

Example

- Now we have <expr> <op> <expr>
- The first NT is <expr>
- The remaining chromosome is:

  280 45 127 29 59 163

- The NT <expr> has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
     |   (<expr> <op> <expr>)
     |   <fun> (<expr>)
     |   <var>
```

- Using the next codon 45, we get $45 \mod 4 = 1$
- Thus, we will use (<expr> <op> <expr>)

Example

- Now we have (<expr> <op> <expr>) <op> <expr>
- The next NT is <expr>
- The remaining chromosome is:

  280 45 127 29 59 163

- The NT <expr> has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
     |   (<expr> <op> <expr>)
     |   <fun> (<expr>)
     |   <var>
```

- Using the next codon 127, we get $127 \mod 4 = 3$
- Thus, we will use <var>

Example

- Now we have `(<var> <op> <expr>) <op> <expr>`
- The next NT is `<var>`
- The remaining chromosome is:

  280 45 127 29 59 163

- The NT `<var>` has 1 production rule:

  `<var> ::= X`

- Using the next codon 29, we get $29 \mod 1 = 0$
- Thus, we will use `X`

Example

- Now we have `(X <op> <expr>) <op> <expr>`
- The next NT is `<op>`
- We continue using the chromosome:

  280 45 127 29 59 163
- The NT has 4 production rules:

  `<op>  ::= + | - | * | /`
- Using the next codon 59, we get $59 \mod 4 = 3$
- Thus, we will use $*$

Example

- Now we have `(X * <expr>) <op> <expr>`
- The next NT is `<expr>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
     |   (<expr> <op> <expr>)
     |   <fun> (<expr>)
     |   <var>
```

- Using the next codon 163, we get $163 \mod 4 = 3$
- Thus, we will use `<var>`

Example

- Now we have `(X * <var>) <op> <expr>`
- The next NT is `<var>`
- We reuse the chromossome:

  280 45 127 29 59 163
- The NT `<var>` has 1 production rule:

  `<var> ::= X`
- Using the next codon 280, we get $280 \mod 1 = 0$
- Thus, we will use `X`

Example

- Now we have `(X * X) <op> <expr>`
- The next NT is `<op>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

  `<op>  ::= + | - | * | /`

- Using the next codon 45, we get $45 \mod 4 = 1$
- Thus, we will use -

Example

- Now we have `(X * X) - <expr>`
- The next NT is `<expr>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
     |   (<expr> <op> <expr>)
     |   <fun> (<expr>)
     |   <var>
```

- Using the next codon 127, we get 127 mod 4 = 3
- Thus, we will use `<var>`

Example

- Now we have `(X * X) - <var>`
- The next NT is `<var>`
- We continue using the chromosome:

  280 45 127 29 59 163
- The NT has 1 production rule:

  `<var> ::= X`
- Using the next codon 29, we get 29 mod $1 = 0$
- Thus, we will use `<X>`

Example

- Thus, the phenotype corresponding to the chromosome

  280 45 127 29 59 163

- is `(X * X) - X`

Notes

- It is possible that the entire chromossome is not used
- Each of the chromosomes depends on context
- The value 127 would mean selecting:
  - the second value for `<fun>` (it has 3 values)
  - the last value for `<expr>` (it has 4 values)
- The value 115 has the same properties
- Therefore, if 127 is mutated ito 115, it is a *silent mutation*

Silent Mutations

- A value can be mutated to another integer with the same properties
- The mutated value can be non-encoding

Non locality

- A mutation can perform a large change instead of a small one
- This often happens when changing values near the top of the tree

Initialization

- Use **Ramped half and half**
- Individuals are generated in such a way that their derivation trees follow *Ramped half and half*
- For each node of the derivation tree
  - Record which choice was made for each production
  - Generate a number that, following the mod rule, gives the correct production

Genetic operators

- The operators used in the genetic algorithms can be used:
  - One point crossover
  - Point mutation
- New operators can be created, for instance:

  Pruning
  - If a large part of both individuals is non-coding (introns) then crossover will not perform any change
  - Pruning removes the introns by removing the non-coding regions

Parameters

- Same as GA
  - Population size
  - Number of iterations
  - Genome size
  - Genetic operator probabilities
  - Fitness function
- Ways of limiting individuals' complexity:
  - Maximum depth of the derivation tree
  - Maximum number of wrappings

Guidelines for creating a grammar

- Don't add NT with only one production/alternative, e.g., ::= X
- These NTs will use up elements from the genome
- Don't add unnecessary complexity to the grammar
- Parsimony is an important concept, as in GP
- This increases the search space and thus the effort necessary for finding solutions

Structured Grammatical Evolution

- The genotype is a list with a one-to-one correspondence between genes and grammar non-terminals
- SGE avoids locality and redundancy issues by directly linking the genotype to the grammar's expansion process.

SGE Genotype and Mapping

Each non-terminal in the grammar is assigned its own sequence of codons (integers) in the genotype.

The mapping starts with the axiom and recursively expands symbols, reading codons directly tied to each grammar rule.

Grammar recursion is handled by preprocessing; the grammar must specify how many times non-terminals can expand to prevent infinite recursion.

Preprocessing

- If the grammar is recursive, it must be preprocessed
- Grammar is rewritten in order to specify the maximum number of expansions
- New grammar is then used

SGE Example

```
<expr> ::= <expr> <op> <expr>
     |  (<expr> <op> <expr>)
     |  <fun> (<expr>)
     |  <var>
<op>   ::= + | - | * | /
<fun>  ::= Sin | Cos | Tan
<var>  ::= X | Y
```

SGE Example

```
<expr>        <op> <fun>    <var>
[16 22 35]   [100] [39]     [7 6]

<expr> <op> <expr>           ( 16)
<fun> (<expr>) <op> <expr>   ( 22)
Sin (<expr>) <op> <expr>     ( 39)
Sin (X) <op> <expr>          (  7)
Sin (X) + <expr>             (100)
Sin (X) + <var>              ( 35)
Sin (X) + X                  (  6)
```

SGE Crossover

- Use uniform crossover over non-terminal vectors

```
Parents
[16 22 35] [100] [39] [173 6]
[49 34 17] [ 24] [19] [35 19]

Offspring
[49 34 17] [100] [39] [35 19]
[16 22 35] [ 24] [19] [173 6]
```

SGE Mutation

- Change one integer value
- Select new integer value from the possible alternatives

SGE Initialization

- Individuals are always initialized by using vectors that create valid individuals
- There are no non-coding regions

Dynamic Structured Grammatical Evolution (DSGE)

- SGE needs to preprocess the grammar before the run
- DSGE dynamically grows the genotype during the run whenever needed

Differences

SGE

- Uses a fixed structure
- Each gene is linked to a non-terminal
- Requires preprocessing to remove recursive productions to a maximum recursion depth
- Genotype must encode the largest possible derivation tree, even when it is not used

DSGE

- Uses a dynamic structure with the necessary mapping numbers during the evolutionary search.
- No need for preprocessing
- Uses truly dynamic, recursive structures without pre-defined limits
- Efficient genotype that only grows as needed

Probabilistic Structured Grammatical Evolution (PSGE)

- PSGE uses a Probabilistic Context-Free Grammar (PCFG)
- For each expansion step, a random value is used to select the grammar rule, weighted according to the PCFG probabilities
- This probabilistic rule selection enables adaptive and domain-focused search, evolving solutions according to statistical patterns
- The PCFG's probabilities are updated each iteration according to the evolution of the best individuals in the population
- Rules used to generate the fittest individual have their probabilities increased

PSGE Mapping Process

- Recursive procedure
- At each step a codon is sampled randomly and appended to the corresponding non-terminal's codon list.
- The codon's value is used to determine which rule to expand next for the symbol, referencing the PCFG's probability table.
- The process continues until a valid phenotype is constructed or until maximum tree depth is reached.

PSGE Example

```
<expr> ::= <expr> <op> <expr>      [0; 0.33[
         |  (<expr> <op> <expr>)   [0.33; 0.55[
         |  <fun> (<expr>)         [0.55; 0.72[
         |  <var>                  [0.72; 1]
<op>   ::= +                       [0; 0.2[
         | -                       [0.2; 0.3[
         | *                       [0.3; 0.7[
         | /                       [0.7; 1]
<fun>  ::= Sin                     [0; 0.5[
         | Cos                     [0.5; 1]
<var>  ::= X                       [0; 1]
```

<expr>	<op>	<fun>	<var>
[0.2 0.6 0.8 0.75]	[0.45]	[0.2]	[0.7 0.2]

PSGE Example

```
<expr> ::= <expr> <op> <expr>      [0; 0.33[
         |  (<expr> <op> <expr>)   [0.33; 0.55[
         |  <fun> (<expr>)         [0.55; 0.72[
         |  <var>                  [0.72; 1]

<expr> <op> <expr>

<expr>                 <op>     <fun>    <var>
[0.2 0.6 0.8 0.75]   [0.45]    [0.2]    [0.7 0.2]
```

PSGE Example

```
<expr> ::= <expr> <op> <expr>      [0; 0.33[
         |  (<expr> <op> <expr>)   [0.33; 0.55[
         |  <fun> (<expr>)         [0.55; 0.72[
         |  <var>                  [0.72; 1]

<fun>(<expr>) <op> <expr>

<expr>              <op>     <fun>    <var>
[0.6 0.8 0.75]      [0.45]   [0.2]    [0.7 0.2]
```

PSGE Example

```
<fun>  ::= Sin                    [0; 0.5[
         | Cos                    [0.5; 1]

Sin(<expr>) <op> <expr>

<expr>              <op>     <fun>    <var>
[0.8 0.75]          [0.45]   [0.2]    [0.7 0.2]
```

PSGE Example

```
<expr> ::= <expr> <op> <expr>      [0; 0.33[
         |  (<expr> <op> <expr>)   [0.33; 0.55[
         |  <fun> (<expr>)         [0.55; 0.72[
         |  <var>                  [0.72; 1]

Sin(<var>) <op> <expr>

<expr>              <op>     <fun>    <var>
[0.8 0.75]          [0.45]   []       [0.7 0.2]
```

PSGE Example

```
<var>  ::= X                              [0; 1]

Sin(X) <op> <expr>

<expr>                  <op>     <fun>    <var>
[0.75]                  [0.45]   []       [0.7 0.2]
```

PSGE Example

```
<op>    ::= +                    [0; 0.2[
          | -                    [0.2; 0.3[
          | *                    [0.3; 0.7[
          | /                    [0.7; 1]

Sin(X) * <expr>

<expr>              <op>      <fun>     <var>
[0.75]              [0.45]    []        [0.2]
```

PSGE Example

```
<expr> ::= <expr> <op> <expr>      [0; 0.33[
         |  (<expr> <op> <expr>)   [0.33; 0.55[
         |  <fun> (<expr>)         [0.55; 0.72[
         |  <var>                  [0.72; 1]
```

```
Sin(X) * <var>
```

<expr>	<op>	<fun>	<var>
[0.75]	[]	[]	[0.2]

PSGE Example

```
<var>  ::= X                          [0; 1]

Sin(X) * X

<expr>              <op>     <fun>    <var>
[]                  []       []       [0.2]
```

PSGE operators

- Uniform crossover as with SGE
- Gaussian noise mutation to change any of the values

SGE/PSGE Advantages

- SGE reduces redundancy and improves locality
- PSGE uses probabilistic mappings for domain learning