GEM - Grammatical Evolution in Matlab (v0.1)

Erik Hemberg, Michael O'Neill

Natural Computing Research & Applications Group
University College Dublin
Ireland

Technical Report

November 19, 2010

Abstract

GEM is an open source implementation of Grammatical Evolution in Matlab developed at UCD's Natural Computing Research & Applications group. As well as providing the characteristic genotype-phenotype mapper of GE a search algorithm engine is also provided.

Contents

1 Introduction

Grammatical Evolution in Matlab (GEM) was developed at UCD's Natural Computing Research & Applications group[1]. It is an open source implementation of Grammatical Evolution (GE) [O'Neill and Ryan, 2003] released under GNU GPL v3.0, which provides a search engine framework in addition to the genotype-phenotype mapper of GE.

This technical report serves as an introduction to the GEM software providing guidelines on its installation and use. In this first release some simple demonstration problems are provided to assist the user to gain an understanding of the code while reading the accompanying tutorials which are provided in this document and on the code's website [NCRA, 2010].

Following a brief introduction to Grammatical Evolution in Section 2, we describe the design of GEM in Section 3, and an introductory tutorials on its use in Sections 4.

2 Grammatical Evolution

Grammatical Evolution (GE) (e.g., [O'Neill and Ryan, 2003]) is a grammar-based form of Genetic Programming [Poli et al., 2008]. It is inspired by representation in molecular biology and combines this with formal grammars. The GE system is flexible and allows the use of alternative search strategies, whether evolutionary, deterministic or of some other approach. This system also includes the ability to bias the search by changing the grammar used. Since a grammar is used to describe the structures that are generated by GE, editing the grammar modifies the output structures. This constraining power is one of GE's main features. The genotype-phenotype, i.e. input-output mapping means that GE allows search operators to be performed on any representation in the algorithm, e.g. on the genotype (integer or binary chromosomes), as well as on partially generated phenotypes, and on the completely generated derivation trees or phenotypes.

The biological inspiration for GE comes from the generation of a protein from a sequence of DNA, which contains several mappings. A simplified description of the generation of a protein from DNA is described in Tab. 1. In Biology, the genotype, DNA is transcribed to RNA, the RNA is translated to amino acids, the amino acids create proteins, and the proteins generate a phenotype. Analogously, for an individual in GE the genotype, binary string, is transcribed to an integer sequence, the integers are translated to production choices via a grammar, and the phenotype is the sentence generated from the grammar.

2.1 GE Control Flow

In GE the control flow of an EA in Fig. 1 is extended with a genotype-phenotype mapping, this is the same as "decoding" in a GA.

[1] http://ncra.ucd.ie

Tab. 1: Comparison of a generation of a protein and the derivation of a sentence in GE. In Biology the genotype, DNA is transcribed to RNA, the RNA is translated to amino acids, the amino acids create proteins, and the proteins generate a phenotype. Analogously, for an individual in GE the genotype, binary string, is transcribed to an integer sequence, the integers are translated to production choices via a grammar, and the phenotype is the sentence generated from the grammar.

Biology		**Grammatical Evolution**
DNA		Binary string
⇓	*Transcription*	⇓
RNA		Integer sequence
⇓	*Translation*	⇓
Amino Acid		Production choice
⇓		⇓
Protein		Sentence(Program)
⇓		⇓
Phenotypic effect		Evaluated sentence

The canonical GE uses a standard GA as a search engine, with crossover and mutation. The steps in a single iteration of GE are generally:

1. **Initialization** Input in the initial solutions is generated, e.g. uniformly randomly generated integer sequences
2. **Mapping** Mapping via a grammar, e.g. CFG (see Section 2.2).
   (a) **Binary to Integer** (Transcription) Binary to integer translation
   (b) **Integer to String** (Translation) Grammar maps integer value to a sentential form (sequence of symbols).
3. **Evaluation** The individual solutions are evaluated.
4. **Operators** Operations on input, e.g. mutation and crossover
   (a) **Selection** Some individuals from the current population are included in a new population. In tournament selection, a tournament size is chosen, and a number of individuals equal to the tournament size are randomly chosen from the population to compete in the tournament. The individual with the best fitness of the individuals selected for the tournament wins the tournament and is selected.
   (b) **Variation operators** Individuals are modified by some operators, e.g. crossover and mutation. In crossover one point in each parent's genotype is selected. The parts on each side of the point are joined to the opposing part from the other parent. This crossover creates two children consisting of one part from each parent. In mutation each input codon has a uniform probability of changing to a new uniform integer value.

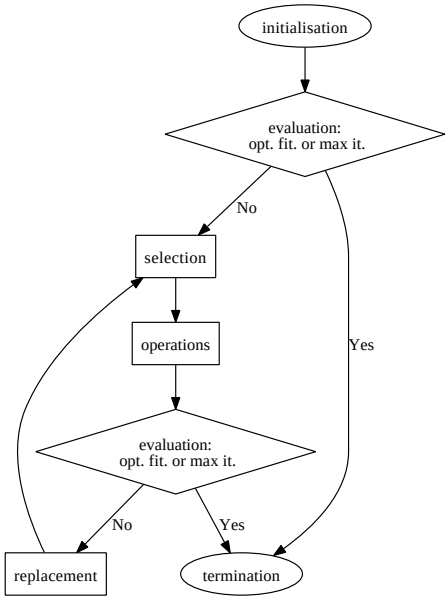

Fig. 1: Flow of a canonical Evolutionary Algorithm. First initialize a population, then evaluate the population, while not optimum found or not max iterations reached: select individual solutions from the population, apply operators to the selected solutions and replace the population.

(c) **Replacement** A new population is created from the selected population and from the current population. If generational replacement is used, the entire population is replaced

5. **Termination** When the start symbol has generated a sentence, the genotype (input) is extended by wrapping. An individual that is not completely mapped, even after wrapping, is called an invalid individual.

These steps complete the algorithm.

2.2 Grammar Mapping in GE

The mapping of GE is shown in Fig. 2. There are different spaces, genotype, phenotype and fitness.

In a Context-Free Grammar the generation of a word is not dependent on the surroundings, see Booth [Booth, 1967]. A CFG is a four tuple $G = \langle N, \Sigma, R, S \rangle$, where:

- N is a finite non-empty set of non-terminal symbols.
- Σ is a finite non-empty set of terminal symbols and $N \cap \Sigma = \emptyset$, the empty set.

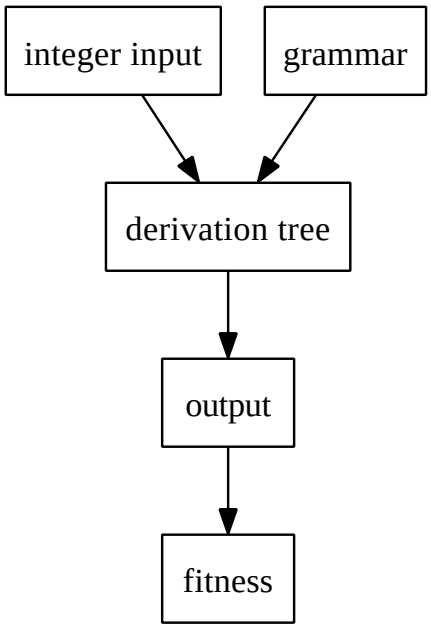

Fig. 2: GE mapping flow: input and grammar are mapped to output that is evaluated and assigned a fitness

- R is a finite set of production rules of the form $R : N \mapsto V^* : A \mapsto \alpha$ or (A, α) where $A \in N$ and $\alpha \in V^*$. V^* is the set of all strings constructed from $N \cup \Sigma$ and $R \subseteq N \times V^*$, $R \neq \emptyset$.
- S is the start symbol, $S \in N$.

"Context-Free" means that for a rule $A \to \alpha$, A can always be replaced by α, regardless of context [Harrison, 1978].

2.2.1 The Grammar

For GE a suitable BNF grammar definition must exist. How much domain knowledge to incorporate is decided by the practitioner, who also defines how general or specific the Backus Naur Form (BNF) grammar is.

In GE, a BNF-grammar describes the output sentences that can be produced by the system, as well as the grammar bias.

The Grammar 1 can be used to generate boolean expressions, and `<expr>` can be transformed into one of three rules. It can become either `( <expr> <biop> <expr> )`, `<uop> <expr>`, or `<bool>`. A grammar can be represented by the tuple $\langle N, \Sigma, R, S \rangle$.

N = { `<expr>, <biop>, <uop>, <bool>` }
Σ = { `and, or, xor, nand, not, true, false, (, )` }
S = { `<expr>` }

```
<expr> ::= ( <expr> <biop> <expr> )
| <uop> <expr>
| <bool>
<biop> ::= and
| or
| xor
| nand
<uop> ::= not
<bool> ::= true
| false
```

Grammar 1: Example of a grammar for boolean expressions. `<expr>` has three production choices, `<biop>` has four production choices, `<uop>` has one production choice and `<bool>` has two production choices.

The code produced after mapping a BNF-grammar in GE will consist of elements of the terminal set Σ. The grammar is used in a generative approach, whereby the evolutionary process evolves the production rules to be applied at each stage of a derivation process, starting from the start symbol, until a complete program is formed. The mapping (derivation) is complete when the sentence is one that is comprised of only elements of Σ.

2.2.2 The Mapping

The genotype is used to map the start symbol into a sentence, by the BNF-grammar. The mapping is done by reading input(*codons*) to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

$$Rule = c \ mod \ r \tag{1}$$

where c is the codon integer value, and r is the number of rule choices for the current non-terminal symbol.

Consider the following rule from the grammar in Grammar 1. Given the non-terminal `<biop>`, which describes the set of boolean operators that can be used, there are four production rules to select from. The choices are labeled from zero.

```
<biop> ::= and          (0)
           | or         (1)
           | xor        (2)
           | nand       (3)
```

If the codon being read produces the integer 6, then Eq. (1) gives $6 \ mod \ 4 = 2$, which would select rule `(2) xor`. In the derivation `<biop>` is replaced with `xor`.

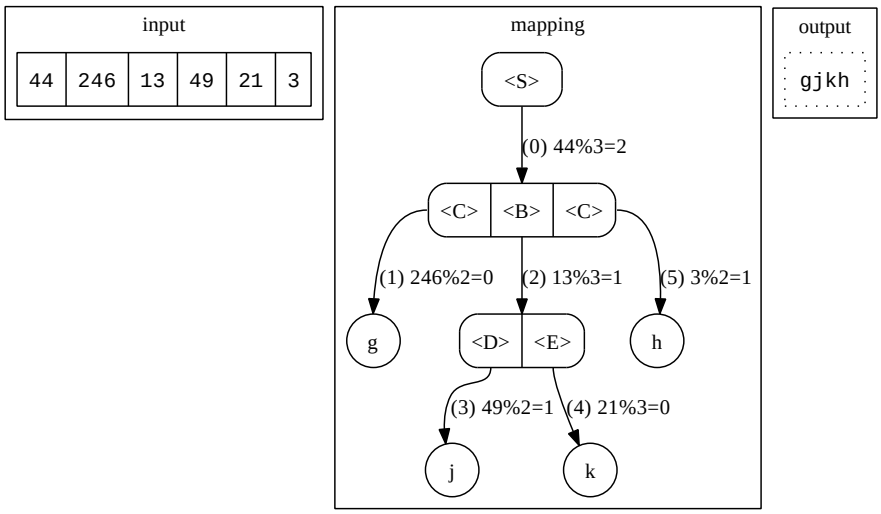

Fig. 3: Example of a derivation tree that generates a word, `gjkh`, using Grammar 2

Each time a production from a rule with more than one production choice has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

The mapping is deterministic, i.e. the same input sequence will map to the same output sequence if the grammar is unchanged, each time the same codon is expressed it will generate the same integer value. But depending on the derivation context, i.e. the current non-terminal to which the codon is being applied, a different production rule may be selected, this is called intrinsic polymorphism [O'Neill et al., 2003].

While the mapping process in GE occurs and a sentence is being built, it can also be represented as a derivation tree. A concrete example of mapping in GE is shown in Fig. 3.

3 Distribution Contents

In the main distribution directory you will see the following files:

`gema.m` the source file

`COPYING` contains the text of the GNU GPL version 3.

`LICENSE` provides the Copyright notice and Licence information for the distribution.

`grammars` contains grammars

`docs` documentation for GEM

```
<S> ::= <C>
| <C><C>
| <C><B><C>
<B> ::= <D>
| <D><E>
| <E>
<C> ::= g
| h
<D> ::= j
| k
<E> ::= k
| l
| m
```

Grammar 2: Example of a grammar for words.

4 Tutorial 0: The Demo Problems

The implemented problem is Symbolic Regression [Koza, 1992]

4.1 Symbolic Regression

The sextic function is used here $x + x^2 + x^3 + x^4 + x^5 + x^6$ with 20 fixed values of $x = [-1.0, -0.9, \ldots, 1.0]$. Fitness is simply the sum of the errors, and an uncomplicated grammar adopted:

```
<expr> ::= ( <expr> <op> <expr> ) | <var>
<op>   ::= + | - | .*
<var> ::= x | 1 | 0
```

Acknowledgments

We would like to thank past and present members of the UCD Natural Computing Research & Applications group. GEM is influenced by ponyGE [Hemberg and McDermott, 2010] and GEVA [O'Neill et al., 2008]. This work is done under Science Foundation Ireland Grant No. 08/IN1/I1868.

References

Taylor L. Booth. *Sequential machines and automata theory.* Wiley, 1967.

MA Harrison. *Introduction to formal language theory.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1978.

Erik Hemberg and James McDermott. pomyGE, November 2010. URL http://code.google.com/p/ponyge/.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992. ISBN 0262111705.

NCRA. NCRA Software, November 2010. URL http://www.ncra.ucd.ie/Site/Software.html.

M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, 2003.

Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2003. ISBN 1402074441.

Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. Geva:grammatical evolution in java. *SIGEVOlution*, 3(2):17–23, Summer 2008.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza).