Integrating Grammatical Evolution with Neural Fitness Functions

Claudio Ferretti and Martina Saletta

Dept. of Informatics, Systems, and Communication (DISCo)

University of Milano-Bicocca

claudio.ferretti@unimib.it

Machine Learning for Source Code Processing

Source code …

given to a neural network

to obtain a classification of single code snippets

Examples:

- detection of identifier misuses
- method names prediction
- classification of programs according to their functionality
- vulnerability detection

Grammatical Evolution (GE)*

Grammar:

```
Rule 1 :   <expr>   ::= <expr> <op> <expr>    (0)
                    |    <var>                (1)

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

Rule 3 :   <var>    ::= x          (0)
                    |    y         (1)
                    |    z         (2)
```

Mapping genotype to phenotype in GE

Genome: | 14 | 7 | 5 | 12 | 3 | 9 |

rule = (codon) MOD (#rules)

Start with Rule 1: <expr>

14 *MOD* 2 = 0 → <expr> <op> <expr>

7 *MOD* 2 = 1 → <var> <op> <expr>

5 *MOD* 3 = *2* → z <op> <expr>

12 *MOD* 4 = 0 → z + <expr>

3 *MOD* 2 = 1 → z + <var>

9 *MOD 3* = 0 → z + x

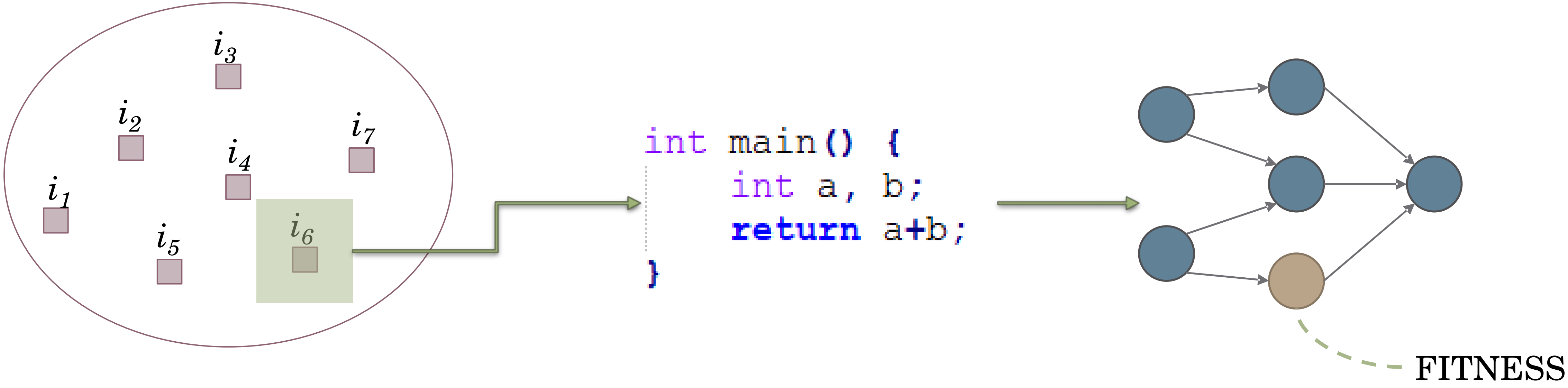
- Genotype: a vector of integers that encodes the productions of a given formal grammar expressed in Backus-Naur Form
- Phenotype: a program in a given language

*M. O'Neill and C. Ryan. *Grammatical evolution*. In: IEEE Transactions on Evolutionary Computation, vol. 5, no. 4, pp. 349-358, 2001

Neural Fitness Functions

- The fitness of each individual is computed as the output of a neuron in a neural network given the individual as input instance

Population **Phenotype** **Neural network**

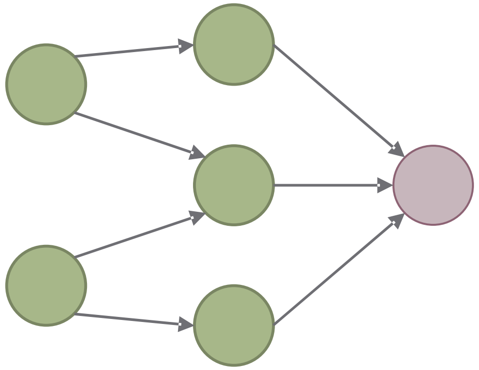


Example 1: binary strings

*see repository: `https://github.com/Martisal/adversarialGE`

```
<E> ::= <A>          <A> ::= 0
    |<E><A>              |<A><B>
    |<E><B>
    |<B>             <B> ::= 1
                         |11
                         |111
```

(Contrived) formal grammar that defines a language of binary strings



Multilayer perceptron successfully trained (regression) in counting the number of zeros of the binary string given as input

We use the output of the MLP for guiding the GE evolution of strings composed of (almost) only zeros

Example 2: deceiving source code classifiers[1]

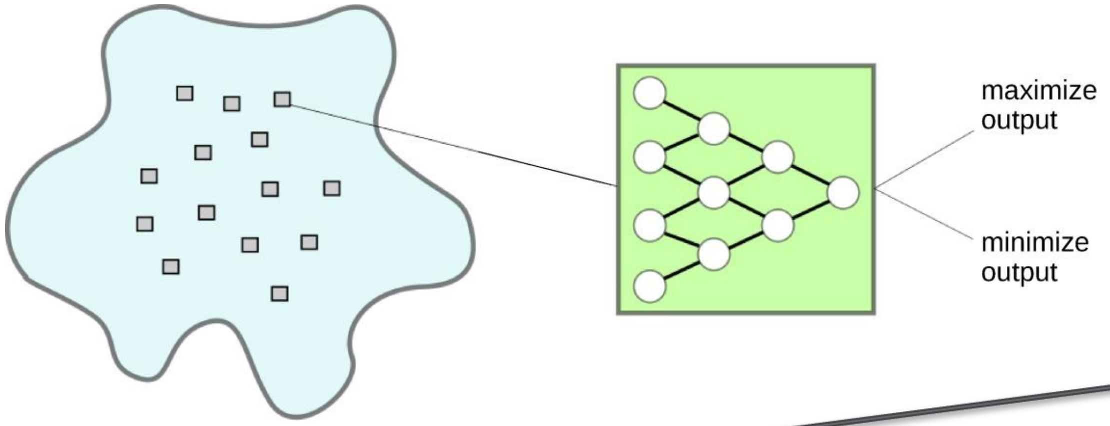
```
<function-definition> ::= <type-specifier> <fdeclarator> { <statements> return <operation>;}
                        | void <fdeclarator> { <statements> return;}
                        | int main(int argc, char **argv) {<statements> return <operation>;}

<operation> ::= <primary-expression>
              | <primary-expression> <operator> <primary-expression>
              | <primary-expression> <operator> <primary-expression> <operator> <primary-expression>

<statement> ::= <type-specifier> <declarator> = <operation>;
              | <type-specifier> <declarator>[<digits>];
              | <declarator> = <operation>;
              | <identifier> -> <identifier> = <operation>;
              | <selection-statement>
              | <iteration-statement>
              | <custom-statement>

<selection-statement> ::= if (<boolean-expression>) {<statements>}
                        | if (<boolean-expression>) {<statements>} else {<statements>}

<parameter-list> ::= <type-specifier> <declarator>
                   | <parameter-list>, <type-specifier> <declarator>

<iteration-statement> ::= while (<boolean-expression>) {<statements>}
                        | do {<statements>} while (<boolean-expression>);

<fdeclarator> ::= <identifier>(<parameter-list>) | <identifier>()

<identifier> ::= str | buf | first | num | id1 | id2 | id3 | id4

<declarator> ::= <identifier> | <pointers><identifier>

<custom-statement> ::= gets(<identifier>);
                     | puts(<identifier>);
                     | strcpy(<identifier>, <identifier>);
                     | strncpy(<identifier>, <identifier>, <digits>);

<constant> ::= <integer-constant> | <character-constant>

<character-constant> ::= 'a'|'b'|'c'|'d'|'e'

<statements> ::= <statement> | <statements> <statement>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<boolean-expression> ::= <operation> >= <operation>
                       | <operation> <= <operation>
                       | <operation> == <operation>
                       | <operation> != <operation>

<integer-constant> ::= <digits> | -<digits>

<digits> ::= <digit> | <digits><digit>

<type-specifier> ::= char | int

<primary-expression> ::= <identifier>
                       | <constant>
                       | <identifier>[<digits>]
                       | <identifier> -> <identifier>
                       | argv[<digits>]

<pointers> ::= * | **

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

Through a simplified C grammar, we use GE for synthesising programs that maximise (or minimize) the output of a model[2] that detects and classifies software vulnerabilities.

[1] C. Ferretti and M. Saletta. *Deceiving neural source code classifiers: finding adversarial examples with grammatical evolution*. In: GECCO21, companion volume. pp. 1889-1897. 2021

[2] Rebecca L. Russell et al. *Automated Vulnerability Detection in Source Code Using Deep Representation Learning.* In: Proceedings of 17° IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 757-762. 2018.

Example 2:
deceiving source code classifiers



Evolution of individuals having an arbitrary classification

Evolution fo individuals able to change the classification of a given input instance but not its functionality