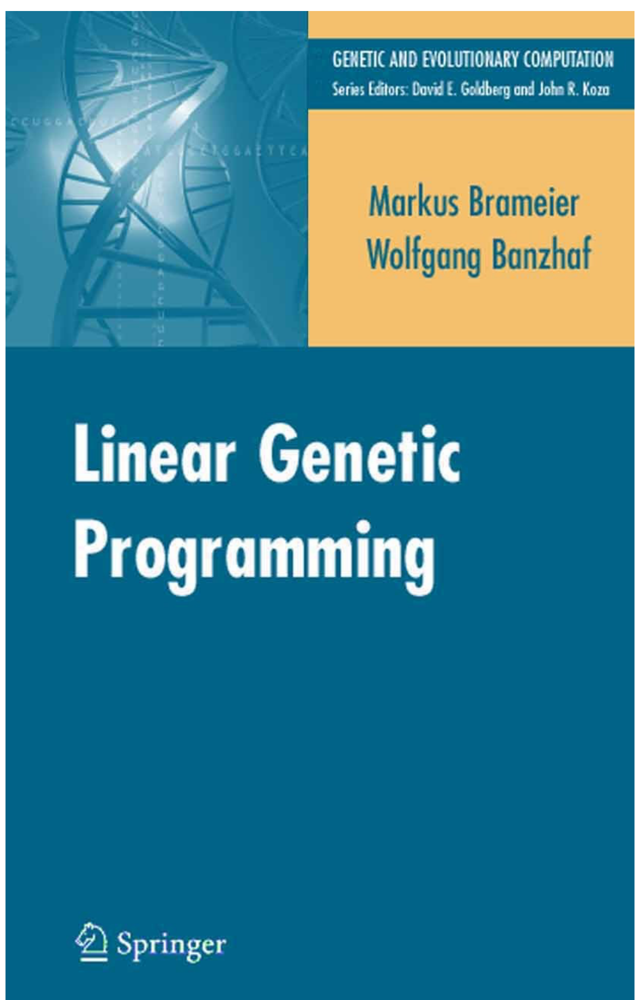
17: Linear Genetic Programming 1



- LGP vs. TGP
- Representation of programs
- Execution of programs
- Non-effective vs. effective code
- Reference book chapters 2, 3
- Reference book: *Linear Genetic Programming, Brameier and Banzhaf, Springer*

Prof. Ting Hu, School of Computing, Queen's University

Genetic Programming variants

- Tree GP
  - Functional expressions represented by syntax trees
  - Inner nodes are functions and leaves hold inputs and constants
- Linear GP
  - Sequence of instructions from an imperative programming language
  - Instructions are operations that accept constants or memory variables (registers) and assign the result to another register
- Others
  - Parallel Distributed GP, Cartesian-GP, push-GP, microGP, Grammatical Evolution…

LGP brief overview

- “Linear”
  - Refers to imperative program representation
  - Represented solutions are highly non-linear
- Early developments
  - Sequential program representation [Cramer, 1985]
  - General linear programs [Banzhaf, 1993]
  - Evolution of machine code (binary) [Nordin, 1994]
  - Aim: independent from any imperative programming language
- Application areas
  - Symbolic regression, classification, …

Motivations

- Investigation of different representations
  - Applications in different problem domains
  - No Free Lunch!

- Major differences comparing to Tree-GP
  - Smaller variation sizes
  - Higher variability -> more compact representation
  - More efficient execution (especially for machine-code LGP no interpretation is needed!)
  - Noneffective code segments coexist with effective code

LGP fundamentals - Representation

- von Neumann Architecture
  - Four major subsystems:
    - memory, input/output, arithmetic/logic unit, control unit
  - Stored program concept
  - ***Sequential*** execution of instructions

- Imperative instruction includes
  - an operation on operand (source) registers
  - an assignment of the result to a destination register
  - 2-register instruction, e.g. $r_i = sin(r_j)$ or $r_i = r_i + r_j$
  - 3-register instruction

- LGP program is a variable-length sequence of instructions

LGP fundamentals - Representation

- Instructions
  - Constant registers, usually explicitly defined and remain fixed (read-only)

    proportion of constants in the program p_{const}
  - Register set: user-defined number of variable registers

    input registers, calculation registers, output register(s) r_0
  - Sufficient number of registers is important

LGP instruction types:

Instruction type	General notation	Input range
Arithmetic operations	$r_i := r_j + r_k$ $r_i := r_j - r_k$ $r_i := r_j \times r_k$ $r_i := r_j \,/\, r_k$	$r_i, r_j, r_k \in \mathbb{R}$
Exponential functions	$r_i := {r_j}^{(r_k)}$ $r_i := e^{r_j}$ $r_i := ln(r_j)$ $r_i := {r_j}^2$ $r_i := \sqrt{r_j}$	$r_i, r_j, r_k \in \mathbb{R}$
Trigonomic functions	$r_i := sin(r_j)$ $r_i := cos(r_j)$	$r_i, r_j, r_k \in \mathbb{R}$
Boolean operations	$r_i := r_j \wedge r_k$ $r_i := r_j \vee r_k$ $r_i := \neg\, r_j$	$r_i, r_j, r_k \in \mathbb{B}$
Conditional branches	$if\ (r_j > r_k)$ $if\ (r_j \leq r_k)$ $if\ (r_j)$	$r_j, r_k \in \mathbb{R}$ $r_j \in \mathbb{B}$

Execution of a program

- Sequential execution of instructions (unless specified)
    - Variable registers will take input variable values
    - Calculation registers, including the output registers, are initialized with constants
    - Constant registers are initialized and then read-only
    - Variable registered are read-only
    - Final value stored in the output register is the program output

Execution of a program - an example

r_1, r_2 take input variables x_1, x_2

r_0, r_3 are the calculation registers, and r_0 is the output register

I1: $r_0 = r_2 + 5$

I2: $r_3 = r_1 \times 3$

I3: $r_3 = r_3 - 1$

I4: $r_0 = r_3 \times r_0$

Non-effective code

- Effective and non-effective (*intron*) code
  - An **instruction** of a linear genetic program is *effective* at its position iff it influences the output of the program for at least one possible input situation
  - A **register** is *effective* for a certain program position iff its manipulation can affect the behavior, i.e., an output, of the program
  - Effective instructions necessarily manipulate effective registers

Structural and semantic intron

- Structural introns and semantic introns
  - **Structural intron** denotes single noneffective instruction that emerge in a linear program from manipulating noneffective registers

    e.g., $r_1 = r_0 \times 5$

    $r_0 = r_0 + 1$
  - **Semantic intron** is a noneffective instruction or a noneffective combination of instructions that manipulate effective register(s)

    e.g., $r_0 = r_0 \times 1$

How to detect structural intron

- Detecting and removing structurally non-effective code

Example:

I1: $r_2 = r_2 + 5$

I2: $r_1 = r_2 \times r_3$

I3: $r_3 = r_1 \times 3$

I4: $r_0 = r_2 - 1$

I5: $r_0 = r_0 \times r_2$

Structural intron removal algorithm

- Detecting and removing structurally non-effective code

ALGORITHM 3.1 (*detection of structural introns*)

1. Let set R_{eff} always contain all registers that are effective at the current program position. $R_{eff} := \{ r \mid r \text{ is output register } \}$.
   Start at the last program instruction and move backwards.

2. Mark the next preceding operation in program with destination register $r_{dest} \in R_{eff}$. If such an instruction is not found then $\rightarrow$ 5.

3. If the operation directly follows a branch or a sequence of branches then mark these instructions too. Otherwise remove r_{dest} from R_{eff}.

4. Insert each source (operand) register r_{op} of newly marked instructions in R_{eff} if not already contained. $\rightarrow$ 2.

5. Stop. All unmarked instructions are introns.

Example again

- Detecting and removing structurally non-effective code

Example:

I1: $r_2 = r_2 + 5$

I2: $r_1 = r_2 \times r_3$

I3: $r_3 = r_1 \times 3$

I4: $r_0 = r_2 - 1$

I5: $r_0 = r_0 \times r_2$

Another example

- Detecting and removing structurally non-effective code

Exercise:

I1: $r_0 = r_1 - 2$

I2: $r_1 = r_2 \times r_0$

I3: $r_1 = r_2 + 3$

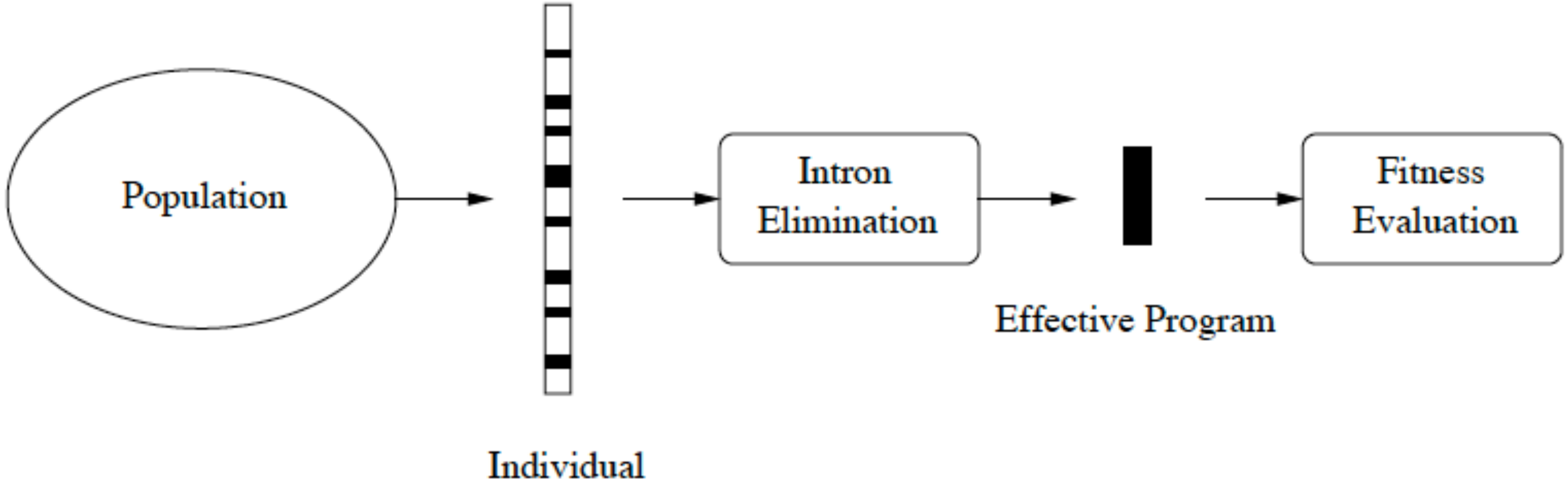
I4: $r_0 = r_2 - 1$

I5: $r_0 = r_1 + r_0$

I6: $r_1 = r_0 \times r_0$

Removing structural intron

- Detecting and removing structurally non-effective code
  - for fitness evaluation only on copies of individuals



How to detect semantic intron

- Detecting and removing semantically non-effective code

ALGORITHM 3.2 (*elimination of semantic introns*)

1. Calculate the fitness $\mathcal{F}_{ref}$ of the program on a set of m data examples (fitness cases) as a reference value. Start at the first program instruction at position $i := 1$.

2. Delete the instruction at the current program position i.

3. Evaluate the program again.

4. If its fitness $\mathcal{F} = \mathcal{F}_{ref}$ then the deleted instruction is an intron. Otherwise, reinsert the instruction at position i.

5. Move to the next instruction at position $i := i + 1$.

6. Stop, if the end of program has been reached. Otherwise $\rightarrow$ 2.