Programming the Basics of an Evolutionary Algorithm(EA)

it3708 - Subsymbolic AI methods

Aleksander Burkow

Department of Computer and Information Science
Norwegian University of Science and Technology

Abstract

In this report, a general purpose EA solver and framework is presented and applied to the One-Max problem. Further, optimal parent selection strategies, reproduction schemes, and parameters for the evolutionary algorithm are discussed.

Contents

List of Figures

Listings

1 Description of EA

The EA program consists of the following modules.

1. A Generic EA solver class. Acts as the evolutionary loop, calling the correct methods on the EA problem.
2. A generic EA problem class. All functions are sent in as parameters, so this class simply acts as the skeleton which ties the hot-swappable components together.
3. Collections of fitness functions, adult selection functions, parent selection functions and genotype to phenotype converters.
4. A testrunner with logging and statistical capabilities.

Composing an EA problem is as simple as selecting the required functions from each set (parent selection etc), bind them together using the generic EA problem class, and then apply it to the EA solver.

2 Code modularity

A EA problem is composed by providing the functions one might want to use to an implementation of the generic EA problem class 1. The Generic EA solver is initialized with the problem, when to terminate the simulation, and then executed.

Listing 1: *constructing an ea problem*

```
ea_problem = EA_PROBLEM(vector_length=VECTOR_LENGTH,
                        fitness_function=FITNESS_FUNCTION,
                        adult_selection_function=
                            ADULT_SELECTION_FUNCTION,
                        # Parameters omitted for brevity
                        n_reproducing_couples=N_REPRODUCING_COUPLES,
                        crossover_chance=CROSSOVER_CHANCE,
                        mutation_chance=MUTATION_CHANCE)

generation, winner = generic_ea_solver\
    .GenericEaSolver(ea_problem,
                     GENERATION_LIMIT).start_simulation()
```

Default constants are defined as follows 2. For each step of the evolutionary chain one can pick and chose between multiple variants. The collection of adult selection functions contains both full generational replacement, generational mixing, and over production, for example.

Overrides for values are accepted at runtime, but omitted for brevity.

Listing 2: *Initial values for the EA problem*

```
EA_PROBLEM = one_max_problem.OneMaxProblem
PHENO_FROM_GENO_FUNCTION = pheno_from_geno_functions\
  .identity_function
FITNESS_FUNCTION = fitness_functions\
  .omf_punish
ADULT_SELECTION_FUNCTION = adult_selection_functions\
  .full_generational_replacement
PARENT_SELECTION_FUNCTION = parent_selection_functions\
  .fitness_proportionate

POPULATION_SIZE = 200
N_REPRODUCING_COUPLES = POPULATION_SIZE / 2
VECTOR_LENGTH = 40
GENERATION_LIMIT = 400

CROSSOVER_CHANCE = 1
MUTATION_CHANCE = 0.001
```

3 40-bit One-Max with FGR and FP

Chosing a good fitness function was paramount in minimizing the needed population size. A punishing one was chosen (fitness proportionate with the inverse of the error), which greatly increased the effectiveness of fitness-proportionate scaling due to greater spacing of good phenotypes.

A population size of 80 was consistently able to terminate in 100 generations or less (52 generations average, standard deviation 12.61, 100 runs sample size).

A per bit mutation chance of $(0.01 > \epsilon > 0.001)$ and a crossover rate of $\epsilon > 0.9$ was found to be optimal. Higher mutation chances led to randomized search, the opposite causing stagnation of the gene pool due to cycles and lack of variance. If none of the genomes have bit n set, no amount of crossover can generate a solution.

References