GPR MANUAL

Contents

[in package MGL-GPR]

- **[system]** **"mgl-gpr"**

```
- _Version:_ 0.0.1
- _Description:_ `mgl-gpr` is a library of evolutionary algorithms such
    as Genetic Programming (evolving typed expressions from a set of
    operators and constants) and Differential Evolution.
- _Licence:_ MIT, see COPYING.
- _Author:_ Gábor Melis <mega@retes.hu>
- _Mailto:_ [mega@retes.hu](mailto:mega@retes.hu)
- _Homepage:_ <http://melisgl.github.io/mgl-gpr>
- _Bug tracker:_ <https://github.com/melisgl/mgl-gpr/issues>
```

```
- _Source control:_ [GIT](https://github.com/melisgl/mgl-gpr.git)
- *Depends on:* alexandria, [mgl-pax][6fdb]
```

1 Links

Here is the official repository and the HTML documentation for the latest version.

2 Background

Evolutionary algorithms are optimization tools that assume little of the task at hand. Often they are population based, that is, there is a set of individuals that each represent a candidate solution. Individuals are combined and changed with crossover and mutationlike operators to produce the next generation. Individuals with lower fitness have a lower probability to survive than those with higher fitness. In this way, the fitness function defines the optimization task.

Typically, EAs are quick to get up and running, can produce reasonable results across a wild variety of domains, but they may need a bit of fiddling to perform well and domain specific approaches will almost always have better results. All in all, EA can be very useful to cut down on the tedium of human trial and error. However, they have serious problems scaling to higher number of variables.

This library grew from the Genetic Programming implementation I wrote while working for Ravenpack who agreed to release it under an MIT licence. Several years later I cleaned it up, and documented it. Enjoy.

3 Evolutionary Algorithms

Evolutionary algorithm is an umbrella term. In this section we first discuss the concepts common to conrete evolutionary algorithms Genetic Programming and Differential Evolution.

- **[class]** **evolutionary-algorithm**

  The `evolutionary-algorithm` is an abstract base class for generational, population based optimization algorithms.

3.1 Populations

The currenly implemented EAs are generational. That is, they maintain a population of candidate solutions (also known as individuals) which they replace with the next generation of individuals.

- **[accessor]** **population-size** *evolutionary-algorithm* *(:population-size)*

  The number of individuals in a generation. This is a very important parameter. Too low and there won't be enough diversity in the population, too high and convergence will be slow.

- **[accessor]** **population** *evolutionary-algorithm (= (make-array 0 :adjustable 0 :fill-pointer t))*

  An adjustable array with a fill-pointer that holds the individuals that make up the population.

- **[reader]** **generation-counter** *evolutionary-algorithm (= 0)*

  A counter that starts from 0 and is incremented by `advance`. All accessors of `evolutionary-algorithm` are allowed to be specialized on a subclass which allows them to be functions of `generation-counter`.

- **[function]** **add-individual** *ea individual*

  Adds `individual` to `population` of `ea`. Usually called when initializing the `ea`.

3.2 Evaluation

- **[reader]** **evaluator** *evolutionary-algorithm (:evaluator)*

  A function of two arguments: the `evolutionary-algorithm` object and an individual. It must return the fitness of the individual. For Genetic Programming, the evaluator often simply calls `eval`, or `compile` + `funcall`, and compares the result to some gold standard. It is also typical to slightly penalize solutions with too many nodes to control complexity and evaluation cost (see `count-nodes`). For Differential Evolution, individuals are conceptually (and often implemented as) vectors of numbers so the fitness function may include an L1 or L2 penalty term.

  Alternatively, one can specify `mass-evaluator` instead.

- **[reader]** **mass-evaluator** *evolutionary-algorithm (:mass-evaluator = nil)*

  `nil` or a function of three arguments: the `evolutionary-algorithm` object, the population vector and the fitness vector into which the fitnesses of the individuals in the population vector shall be written. By specifying `mass-evaluator` instead of an `evaluator`, one can, for example, distribute costly evaluations over multiple threads. `mass-evaluator` has precedence over `evaluator`.

- **[reader]** **fitness-key** *evolutionary-algorithm (:fitness-key = #'identity)*

  A function that returns a real number for an object returned by `evaluator`. It is called when two fitness are to be compared. The default value is `#'identity` which is sufficient when `evaluator` returns real numbers. However, sometimes the evaluator returns more information about the solution (such as fitness in various situations) and `fitness-key` key be used to select the fitness value.

3.3 Training

Training is easy: one creates an object of a subclass of `evolutionary-algorithm` such as `genetic-programming` or `differential-evolution`, creates the initial population by adding individuals to it (see `add-individual`) and calls `advance` in a loop to move on to the next generation until a certain number of generations or until the `fittest` individual is good enough.

- **[generic-function]** **advance** *ea*

  Create the next generation and place it in `population` of `ea`.

- **[reader]** **fittest** *evolutionary-algorithm (= nil)*

  The fittest individual ever to be seen and its fittness as a cons cell.

- **[accessor]** **fittest-changed-fn** *evolutionary-algorithm (:fittest-changed-fn = nil)*

  If non-`nil`, a function that's called when `fittest` is updated with three arguments: the `evolutionary-algorithm` object, the fittest individual and its fitness. Useful for tracking progress.

4 Genetic Programming

4.1 Background

What is Genetic Programming? This is what Wikipedia has to say:

> In artificial intelligence, genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. Essentially GP is a set of instructions and a fitness function to measure how well a computer has performed a task. It is a specialization of genetic algorithms (GA) where each individual is a computer program. It is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

Lisp has a long history of Genetic Programming because GP involves manipulation of expressions which is of course particularly easy with sexps.

4.2 Tutorial

GPR works with typed expressions. Mutation and crossover never produce expressions that fail with a type error. Let's define a couple of operators that work with real numbers and also return a real:

```
(defparameter *operators* (list (operator (+ real real) real)
                                (operator (- real real) real)
                                (operator (* real real) real)
                                (operator (sin real) real)))
```

One cannot build an expression out of these operators because they all have at least one argument. Let's define some literal classes too. The first is produces random numbers, the second always returns the symbol `*x*`:

```
(defparameter *literals* (list (literal (real)
                                 (- (random 32.0) 16.0))
                               (literal (real)
                                 '*x*)))
```

Armed with `*operators*` and `*literals*`, one can already build random expressions with `random-expression`, but we also need to define how good a certain expression is which is called *fitness*.

In this example, we are going to perform symbolic regression, that is, try to find an expression that approximates some target expression well:

```
(defparameter *target-expr* '(+ 7 (sin (expt (* *x* 2 pi) 2))))
```

Think of `*target-expr*` as a function of `*x*`. The evaluator function will bind the special `*x*` to the input and simply `eval` the expression to be evaluated.

```
(defvar *x*)
```

The evaluator function calculates the average difference between `expr` and `target-expr`, penalizes large expressions and returns the fitness of `expr`. Expressions with higher fitness have higher chance to produce offsprings.

```
(defun evaluate (gp expr target-expr)
  (declare (ignore gp))
  (/ 1
     (1+
      ;; Calculate average difference from target.
      (/ (loop for x from 0d0 to 10d0 by 0.5d0
               summing (let ((*x* x))
                         (abs (- (eval expr)
                                 (eval target-expr)))))
         21))
     ;; Penalize large expressions.
     (let ((min-penalized-size 40)
           (size (count-nodes expr)))
       (if (< size min-penalized-size)
           1
           (exp (min 120 (/ (- size min-penalized-size) 10d0)))))))
```

When an expression is to undergo mutation, a randomizer function is called. Here we change literal numbers slightly, or produce an entirely new random expression that will be substituted for `expr`:

```
(defun randomize (gp type expr)
  (if (and (numberp expr)
           (< (random 1.0) 0.5))
      (+ expr (random 1.0) -0.5)
      (random-gp-expression gp (lambda (level)
                                 (<= 3 level))
                            :type type)))
```

That's about it. Now we create a GP instance hooking everything up, set up the initial population and just call `advance` a couple of times to create new generations of expressions.

```
(defun run ()
  (let ((*print-length* nil)
        (*print-level* nil)
```

```
      (gp (make-instance
           'gp
           :toplevel-type 'real
           :operators *operators*
           :literals *literals*
           :population-size 1000
           :copy-chance 0.0
           :mutation-chance 0.5
           :evaluator (lambda (gp expr)
                        (evaluate gp expr *target-expr*))
           :randomizer 'randomize
           :selector (lambda (gp fitnesses)
                       (declare (ignore gp))
                       (hold-tournament fitnesses :n-contestants 2))
           :fittest-changed-fn
           (lambda (gp fittest fitness)
             (format t "Best fitness until generation ~S: ~S for~%  ~S~%"
                     (generation-counter gp) fitness fittest)))))
  (loop repeat (population-size gp) do
    (add-individual gp (random-gp-expression gp (lambda (level)
                                                  (<= 5 level)))))
  (loop repeat 1000 do
    (when (zerop (mod (generation-counter gp) 20))
      (format t "Generation ~S~%" (generation-counter gp)))
    (advance gp))
  (destructuring-bind (fittest . fitness) (fittest gp)
    (format t "Best fitness: ~S for~%  ~S~%" fitness fittest))))
```

Note that this example can be found in example/symbolic-regression.lisp.

4.3 Expressions

Genetic programming works with a population of individuals. The individuals are sexps that may be evaluated directly by `eval` or by other means. The internal nodes and the leafs of the sexp as a tree represent the application of operators and literal objects, respectively. Note that currently there is no way to represent literal lists.

- **[class]** **expression-class**

  An object of `expression-class` defines two things: how to build a random expression that belongs to that expression class and what lisp type those expressions evaluate to.

- **[reader]** **result-type** *expression-class (:result-type)*

  Expressions belonging to this expression class must evaluate to a value of this lisp type.

- **[reader]** **weight** *expression-class (:weight = 1)*

  The probability of an expression class to be selected from a set of candidates is proportional to its weight.

- **[class]** **operator** *expression-class*

Defines how the symbol `name` in the function position of a list can be combined arguments: how many and of what types. The following defines `+` as an operator that adds two `float`(`0` `1`)s:

```
(make-instance 'operator
               :name '+
               :result-type float
               :argument-types '(float float))
```

See the macro `operator` for a shorthand for the above.

Currently no lambda list keywords are supported and there is no way to define how an expression with a particular operator is to be built. See `random-expression`.

- **[reader]** **name** *operator (:name)*

  A symbol that's the name of the operator.

- **[reader]** **argument-types** *operator (:argument-types)*

  A list of lisp types. One for each argument of this operator.

- **[macro]** **operator** *(name &rest arg-types) result-type &key (weight 1)*

  Syntactic sugar for instantiating operators. The example given for `operator` could be written as:

  ```
  (operator (+ float float) float)
  ```

  See `weight` for what `weight` means.

- **[class]** **literal** *expression-class*

  This is slightly misnamed. An object belonging to the `literal` class is not a literal itself, it's a factory for literals via its `builder` function. For example, the following literal builds bytes:

  ```
  (make-instance 'literal
                 :result-type '(unsigned-byte 8)
                 :builder (lambda () (random 256)))
  ```

  In practice, one rarely writes it out like that, because the `literal` macro provides a more convenient shorthand.

- **[reader]** **builder** *literal (:builder)*

  A function of no arguments that returns a random literal that belongs to its literal class.

- **[macro]** **literal** *(result-type &key (weight 1)) &body body*

  Syntactic sugar for defining literal classes. The example given for `literal` could be written as:

  ```
  (literal ((unsigned-byte 8))
    (random 256))
  ```

See `weight` for what `weight` means.

- **[function]** **random-expression** *operators literals type terminate-fn*

  Return an expression built from `operators` and `literals` that evaluates to values of `type`. `terminate-fn` is a function of one argument: the level of the root of the subexpression to be generated in the context of the entire expression. If it returns `t` then a `literal` will be inserted (by calling its `builder` function), else an `operator` with all its necessary arguments.

  The algorithm recursively generates the expression starting from level 0 where only operators and literals with a `result-type` that's a subtype of `type` are considered and one is selected with the unnormalized probability given by its `weight`. On lower levels, the `argument-types` specification of operators is similarly satisfied and the resulting expression should evaluate without without a type error.

  The building of expressions cannot backtrack. If it finds itself in a situation where no literals or operators of the right type are available then it will fail with an error.

4.4 Basics

To start the evolutionary process one creates a GP object, adds to it the individuals (see `add-individual`) that make up the initial population and calls `advance` in a loop to move on to the next generation.

- **[class]** **genetic-programming** *evolutionary-algorithm*

  The `genetic-programming` class defines the search space, how mutation and recombination occur, and hold various parameters of the evolutionary process and the individuals themselves.

- **[function]** **random-gp-expression** *gp terminate-fn &key (type (toplevel-type gp))*

  Creating the initial population by hand is tedious. This convenience function calls `random-expression` to create a random individual that produces `gp`'s `toplevel-type`. By passing in another `type` one can create expressions that fit somewhere else in a larger expression, which is useful in a `randomizer` function.

4.5 Search Space

The search space of the GP is defined by the available operators, literals and the type of the final result produced. The evaluator function acts as the guiding light.

- **[reader]** **operators** *genetic-programming (:operators)*

  The set of `operator`s from which (together with `literal`s) individuals are built.

- **[reader]** **literals** *genetic-programming (:literals)*

  The set of `literal`s from which (together with `operator`s) individuals are built.

- **[reader]** **toplevel-type** *genetic-programming (:toplevel-type = t)*

The type of the results produced by individuals. If the problem is to find the minimum a 1d real function then this may be the symbol `real`. If the problem is to find the shortest route, then this may be a vector. It all depends on the representation of the problem, the operators and the literals.

- **[function]** **count-nodes** *tree &key internal*

  Count the nodes in the sexp `tree`. If `internal` then don't count the leaves.

4.6 Reproduction

The `randomizer` and `selector` functions define how mutation and recombination occur.

- **[reader]** **randomizer** *genetic-programming (:randomizer)*

  Used for mutations, this is a function of three arguments: the GP object, the type the expression must produce and current expression to be replaced with the returned value. It is called with subexpressions of individuals.

- **[reader]** **selector** *genetic-programming (:selector)*

  A function of two arguments: the GP object and a vector of fitnesses. It must return the and index into the fitness vector. The individual whose fitness was thus selected will be selected for reproduction be it copying, mutation or crossover. Typically, this defers to `hold-tournament`.

- **[function]** **hold-tournament** *fitnesses &key select-contestant-fn n-contestants key*

  Select `n-contestants` (all different) for the tournament randomly, represented by indices into `fitnesses` and return the one with the highest fitness. If `select-contestant-fn` is `nil` then contestants are selected randomly with uniform probability. If `select-contestant-fn` is a function, then it's called with `fitnesses` to return an index (that may or may not be already selected for the tournament). Specifying `select-contestant-fn` allows one to conduct 'local' tournaments biased towards a particular region of the index range. `key` is `nil` or a function that select the real fitness value from elements of `fitnesses`.

4.7 Environment

The new generation is created by applying a reproduction operator until `population-size` is reached in the new generation. At each step, a reproduction operator is randomly chosen.

- **[accessor]** **copy-chance** *genetic-programming (:copy-chance = 0)*

  The probability of the copying reproduction operator being chosen. Copying simply creates an exact copy of a single individual.

- **[accessor]** **mutation-chance** *genetic-programming (:mutation-chance = 0)*

  The probability of the mutation reproduction operator being chosen. Mutation creates a randomly altered copy of an individual. See `randomizer`.

If neither copying nor mutation were chosen, then a crossover will take place.

- **[accessor]** **keep-fittest-p** *genetic-programming (:keep-fittest-p = t)*

  If true, then the fittest individual is always copied without mutation to the next generation. Of course, it may also have other offsprings.

5 Differential Evolution

The concepts in this section are covered by Differential Evolution: A Survey of the State-of-the-Art.

- **[class]** **differential-evolution** *evolutionary-algorithm*

  Differential evolution (DE) is an evolutionary algorithm in which individuals are represented by vectors of numbers. New individuals are created by taking linear combinations or by randomly swapping some of these numbers between two individuals.

- **[reader]** **map-weights-into-fn** *differential-evolution (:map-weights-into-fn = #'map-into)*

  The vector of numbers (the 'weights') are most often stored in some kind of array. All individuals must have the same number of weights, but the actual representation can be anything as long as the function in this slot mimics the semantics of `map-into` that's the default.

- **[reader]** **create-individual-fn** *differential-evolution (:create-individual-fn)*

  Holds a function of one argument, the DE, that returns a new individual that needs not be initialized in any way. Typically this just calls `make-array`.

- **[reader]** **mutate-fn** *differential-evolution (:mutate-fn)*

  One of the supplied mutation functions: `mutate/rand/1` `mutate/best/1` `mutate/current-to-best/2`.

- **[reader]** **crossover-fn** *differential-evolution (:crossover-fn = #'crossover/binary)*

  A function of three arguments, the DE and two individuals, that destructively modifies the second individual by using some parts of the first one. Currently, the implemented crossover function is `crossover/binary`.

- **[function]** **mutate/rand/1** *de current best population nursery &key (f 0.5)*

- **[function]** **mutate/best/1** *de current best population nursery &key (f 0.5)*

- **[function]** **mutate/current-to-best/2** *de current best population nursery &key (f 0.5)*

- **[function]** **crossover/binary** *de individual-1 individual-2 &key (crossover-rate 0.5)*

  Destructively modify `individual-2` by replacement each element with a probability of 1 - `crossover-rate` with the corresponding element in `individual-1`. At least one, element is changed. Return `individual-2`.

- **[function]** **select-distinct-random-numbers** *taboos n limit*

5.1 SANSDE

See the paper Self-adaptive Differential Evolution with Neighborhood Search.

- **[class]** **sansde** *differential-evolution*

  SaNSDE is a special DE that dynamically adjust the crossover and mutation are performed. The only parameters are the generic EA ones: `population-size`, `evaluator`, etc. One also has to specify `map-weights-into-fn` and `create-individual-fn`.

6 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: `DEFTYPE`s, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

6.1 Function and Macro Index

6.2 Type Index

6.3 Misc Index