microgp

Release 4!1.0a0.dev24

Apr 21, 2022

Introduction

This is MicroGP4 v1.0_24 pre-alpha “Kiwi” on Thu Apr 21 13:40:10 2022

genindex

Truth emerges more readily from error than from confusion.
— Francis Bacon (1561–1626)

CHAPTER 1

What is MicroGP?

MicroGP is an evolutionary optimizer: given a problem, it first creates a set of random solutions, then iteratively refines and enhances them using the result of their evaluations together with structural information. MicroGP is extremely versatile: it is able to tackle problem those solutions are simple fixed-length bit strings, as well as to optimize realistic assembly programs including loops, interrupts and recursive sub routines; moreover, candidate solutions can be evaluated using proprietary external tools.

MicroGP routinely outperforms both human experts and conventional heuristics, and over the years has been exploited as a coverage-driven fuzzer, as a general-purpose optimizer, and as a framework for prototyping and testing new ideas. Several papers discussing possible applications can be found in the scientific literature.

1.1 Audience

The expected audience for MicroGP4 includes computer scientists, engineers and practitioners.

- MicroGP4 is available as a PyPi package and it can be easily installed using pip.
- MicroGP4 can be used interactively from a Jupyter notebook — just like scikit-learn and keras — allowing to quickly assess new ideas and build prototypes.
- An external tool can be used as evaluator, allowing to handle virtually any problem.
- The evolutionary code can be tweaked injecting user-defined Python functions, without the need to hack the package itself.
- The modular design allows scholars to exploit MicroGP4 for testing new ideas and novel algorithms.

In the past, due to its ability to handle realistic assembly programs and to exploit commercial simulators, MicroGP has been extensively used for the test, verification and validation of programmable cores and sequential circuits.

1.2 History

The MicroGP project started around the year 2000, with the creation of a collection of scripts for optimizing assembly-language programs. A fully working program was first coded in C in 2002, and then re-engineered in C++ in 2006; the design of a Python version started in 2018. MicroGP4 would not have been possible without the help and support of several people.

1.3 Alternatives

Among the remarkable alternatives to MicroGP are:

- ECJ — A Java-based Evolutionary Computation Research System
- DEAP — Distributed Evolutionary Algorithms in Python
- inspyred — A framework for creating bio-inspired computational intelligence algorithms in Python
- Jenetics — A Genetic Algorithm, Evolutionary Algorithm, Genetic Programming, and Multi-objective Optimization library, written in modern-day Java
- Open BEAGLE — A generic C++ framework for evolutionary computation

CHAPTER 2

Installation

MicroGP4 is available as a PyPi package from https://pypi.org/project/microgp/ and installing it is as simple as

```
pip install microgp
```

and then

```
>>> import microgp as ugp4
>>> ugp4.show_banner()
```

Caveat: on some systems the package manager is `pip3`.

2.1 Optional dependencies

The packages coloredlogs, matplotlib and psutil are optional, they will not be installed by default, but are exploited if present.

```
pip install coloredlogs matplotlib psutil
```

- Under Ubuntu/Debian, you may need `Python.h`. For example:

```
sudo apt install python3-dev
pip3 install coloredlogs matplotlib psutil
```

- Under Windows, and if you are using conda, you should probably:

```
conda install coloredlogs matplotlib
conda install --channel conda-forge psutil
```

2.2 Source Code

The source code is hosted on GitHub at https://github.com/squillero/microgp4, the default branch is *usually* aligned with the PyPi package. On Ubuntu/Debian it could be enough to:

```
git clone https://github.com/squillero/microgp4.git
cd microgp4/src
pip3 install -U -r requirements.txt
python3 ./setup.py install
```

CHAPTER 3

Example: One Max

We will consider the OneMax problem as an example of a simple problem solvable with Genetic Algorithms. Given a sequence *N* of random bits composing a word (i.e. *10001010* 8 bits), the fitness score is given based on the number of *ones* present (higher is better). The algorithm must generate a random set of individuals (strings of bits), evolve them till they will contain only ones.

3.1 Base

In version one an individual is composed by a `word_section` which contains a single macro (`word_macro`) with a parameter (`bit`) of type *microgp.parameter.bitstring.Bitstring* of length 8 bits. The main section contains a simple prologue and epilogue.

The evaluator in both versions is a Python method (`evaluator_function`) returning an `int` value that is the sum of *1* in the individual's phenotype.

```
import argparse
import sys

import microgp as ugp
from microgp.utils import logging

if __name__ == "__main__":
    ugp.banner()
    parser = argparse.ArgumentParser()
    parser.add_argument("-v", "--verbose", action="count", default=0, help="increase␣
↪log verbosity")
    parser.add_argument("-d", "--debug", action="store_const", dest="verbose",␣
↪const=2,
                        help="log debug messages (same as -vv)")
    args = parser.parse_args()
    if args.verbose == 0:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.INFO)
    elif args.verbose == 1:
```

(continues on next page)

(continued from previous page)

```
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.VERBOSE)
    elif args.verbose > 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.DEBUG)
        ugp.logging.debug("Verbose level set to DEBUG")
    ugp.logging.cpu_info("Program started")


    # Define a parameter of type ugp.parameter.Bitstring and length = 8
    word8 = ugp.make_parameter(ugp.parameter.Bitstring, len_=8)
    # Define a macro that contains a parameter of type ugp.parameter.Bitstring
    word_macro = ugp.Macro("{word8}", {'word8': word8})
    # Create a section containing a macro
    word_section = ugp.make_section(word_macro, size=(1, 1), name='word_sec')

    # Create a constraints library
    library = ugp.Constraints()
    # Define the sections in the library
    library['main'] = ["Bitstring:", word_section]

    # Define the evaluator method and the fitness type
    def evaluator_function(data: str):
        count = data.count('1')
        return list(str(count))

    library.evaluator = ugp.fitness.make_evaluator(evaluator=evaluator_function,␣
↪fitness_type=ugp.fitness.Lexicographic)

    # Create a list of operators with their aritiy
    operators = ugp.Operators()
    # Add initialization operators
    operators += ugp.GenOperator(ugp.create_random_individual, 0)
    # Add mutation operators
    operators += ugp.GenOperator(ugp.hierarchical_mutation, 1)
    operators += ugp.GenOperator(ugp.flat_mutation, 1)

    # Create the object that will manage the evolution
    mu = 10
    nu = 20
    sigma = 0.7
    lambda_ = 7
    max_age = 10

    darwin = ugp.Darwin(
        constraints=library,
        operators=operators,
        mu=mu,
        nu=nu,
        lambda_=lambda_,
        sigma=sigma,
        max_age=max_age,
    )

    # Evolve and print individuals in population
    darwin.evolve()
    logging.bare("This is the final population:")
    for individual in darwin.population:
        msg = f"Solution {str(individual.node_id)} "
```

(continues on next page)

(continued from previous page)

```
        ugp.print_individual(individual, msg=msg, plot=True)
        ugp.logging.bare(f"Fitness: {individual.fitness}")
        ugp.logging.bare("")

    # Print best individuals
    ugp.print_individual(darwin.archive.individuals, msg="These are the best ever␣
↪individuals:", plot=True)

    ugp.logging.cpu_info("Program completed")
    sys.exit(0)
```

3.2 Structured

In version two an individual is composed by a `word_section` which contains exactly 8 macros (`word_macro`) with a parameter (`bit`) of type *`microgp.parameter.categorical.Categorical`* that can assume as value: 1 or 0. The main section contains a simple prologue and epilogue.

```
import argparse
import sys

import microgp as ugp
from microgp.utils import logging

if __name__ == "__main__":
    ugp.banner()
    parser = argparse.ArgumentParser()
    parser.add_argument("-v", "--verbose", action="count", default=0, help="increase␣
↪log verbosity")
    parser.add_argument("-d", "--debug", action="store_const", dest="verbose",␣
↪const=2,
                        help="log debug messages (same as -vv)")
    args = parser.parse_args()
    if args.verbose == 0:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.INFO)
    elif args.verbose == 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.VERBOSE)
    elif args.verbose > 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.DEBUG)
        ugp.logging.debug("Verbose level set to DEBUG")
    ugp.logging.cpu_info("Program started")

    # Define a parameter of type ugp.parameter.Categorical that can take two values:␣
↪0 or 1
    bit = ugp.make_parameter(ugp.parameter.Categorical, alternatives=[0, 1])

    # Define a macro that contains a parameter of type ugp.parameter.Categorical
    word_macro = ugp.Macro("{bit}", {'bit': bit})

    # Create a section containing 8 macros
    word_section = ugp.make_section(word_macro, size=(8, 8), name='word_sec')

    # Create a constraints library
    library = ugp.Constraints()
    library['main'] = ["Bitstring:", word_section]
```

(continues on next page)

(continued from previous page)

```
    # Define the evaluator method and the fitness type
    def evaluator_function(data: str):
        count = data.count('1')
        return list(str(count))
    library.evaluator = ugp.fitness.make_evaluator(evaluator=evaluator_function,␣
↪fitness_type=ugp.fitness.Lexicographic)

    # Create a list of operators with their arity
    operators = ugp.Operators()
    # Add initialization operators
    operators += ugp.GenOperator(ugp.create_random_individual, 0)
    # Add mutation operators
    operators += ugp.GenOperator(ugp.hierarchical_mutation, 1)
    operators += ugp.GenOperator(ugp.flat_mutation, 1)
    # Add crossover operators
    operators += ugp.GenOperator(ugp.macro_pool_one_cut_point_crossover, 2)
    operators += ugp.GenOperator(ugp.macro_pool_uniform_crossover, 2)

    # Create the object that will manage the evolution
    mu = 10
    nu = 20
    sigma = 0.7
    lambda_ = 7
    max_age = 10

    darwin = ugp.Darwin(
        constraints=library,
        operators=operators,
        mu=mu,
        nu=nu,
        lambda_=lambda_,
        sigma=sigma,
        max_age=max_age,
    )

    # Evolve and print individuals in population
    darwin.evolve()
    logging.bare("This is the final population:")
    for individual in darwin.population:
        msg = f"Solution {str(individual.node_id)} "
        ugp.print_individual(individual, msg=msg, plot=True, score=True)

    # Print best individuals
    ugp.print_individual(darwin.archive.individuals, msg="These are the best ever␣
↪individuals:", plot=True)

    ugp.logging.cpu_info("Program completed")
    sys.exit(0)
```

3.3 Assembly

The following code produces assembly code that can be run on x86 processors. The goal is to generate an assembly script that writes in `eax` a binary number with as much as ones (`1`) as possible.

The evaluator is a `.bat` file that generates an `.exe` file in charge of *call* the script and count the number of ones in the returned integer value.

```
@echo off

rem comment

del a.exe
gcc main.o %1

if exist a.exe (
    .\a.exe
) else (
    echo -1
)
```

A possible solution could be:

```
    .file   "solution.c"
    .text
    .globl  _darwin
    .def    _darwin;        .scl    2;      .type   32;     .endef
_darwin:
LFB17:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5

    movl     $-31312, %eax
    movl     $25598, %ebx
    movl     $-24861, %ecx
    movl     $-19236, %edx

    sub %ebx, %edx
    shl $216, %ecx
    jnz n9
    jnc n23
    cmp %ecx, %ecx
    shl $207, %edx
n9:
    jc n22
    xor %ebx, %eax
    jnz n28
    xor %eax, %ebx
    sub %edx, %edx
    jno n15
n15:
    jz n28
    shr $229, %ebx
    sub %ebx, %eax
    jc n23
    cmp %edx, %ebx
    and %ebx, %ecx
    shl $186, %eax
n22:
```

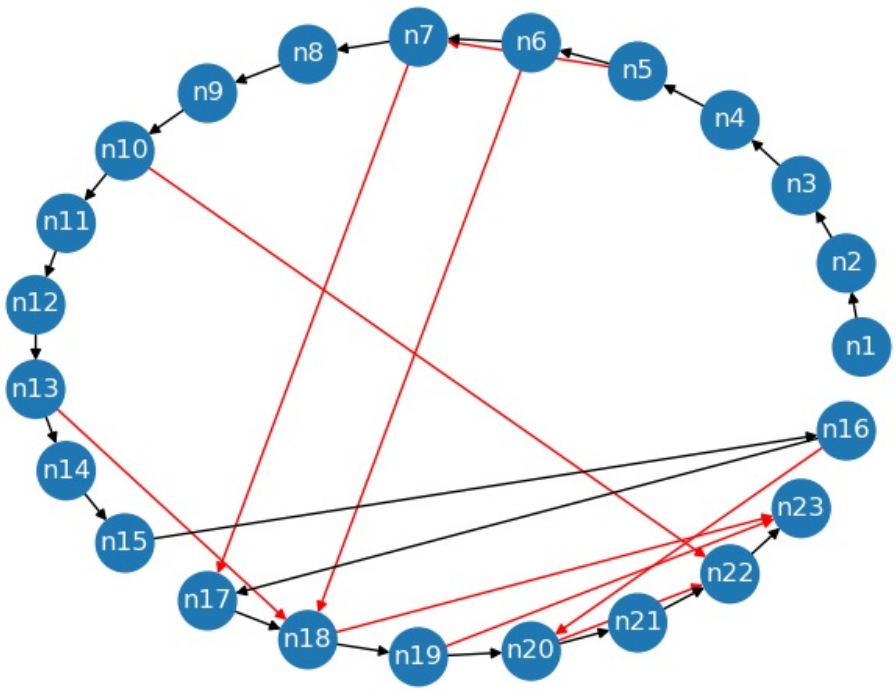
(continues on next page)

(continued from previous page)

```
    cmp %eax, %edx
n23:
    jnz n29
    jz n29
    jmp n28
    jc n29
    shl $143, %ecx
n28:
    or %ebx, %eax
n29:
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE17:
   .ident  "GCC: (MinGW.org GCC-8.2.0-5) 8.2.0"

Fitness score: Lexicographic(29)
```

The correspondent graph_manager plot is:



In the figure the black edges are *next* edges and the red ones are `LocalReferences` (*jump*).

```
import argparse
import sys

import microgp as ugp
```

(continues on next page)

(continued from previous page)

```
from microgp.utils import logging

if __name__ == "__main__":
    ugp.banner()
    parser = argparse.ArgumentParser()
    parser.add_argument("-v", "--verbose", action="count", default=0, help="increase␣
↪log verbosity")
    parser.add_argument("-d", "--debug", action="store_const", dest="verbose",␣
↪const=2,
                        help="log debug messages (same as -vv)")
    args = parser.parse_args()
    if args.verbose == 0:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.INFO)
    elif args.verbose == 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.VERBOSE)
    elif args.verbose > 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.DEBUG)
        ugp.logging.debug("Verbose level set to DEBUG")
    ugp.logging.cpu_info("Program started")

    # Define parameters
    reg_alternatives = ['%eax', '%ebx', '%ecx', '%edx']
    reg_param = ugp.make_parameter(ugp.parameter.Categorical, alternatives=reg_
↪alternatives)
    instr_alternatives = ['add', 'sub', 'and', 'or', 'xor', 'cmp']
    instr_param = ugp.make_parameter(ugp.parameter.Categorical, alternatives=instr_
↪alternatives)
    shift_alternatives = ['shr', 'shl']
    shift_param = ugp.make_parameter(ugp.parameter.Categorical, alternatives=shift_
↪alternatives)
    jmp_alternatives = ['ja', 'jz', 'jnz', 'je', 'jne', 'jc', 'jnc', 'jo', 'jno', 'jmp
↪']
    jmp_instructions = ugp.make_parameter(ugp.parameter.Categorical, alternatives=jmp_
↪alternatives)
    integer = ugp.make_parameter(ugp.parameter.Integer, min=-32768, max=32767)
    int8 = ugp.make_parameter(ugp.parameter.Integer, min=0, max=256)
    jmp_target = ugp.make_parameter(ugp.parameter.LocalReference,
                                    allow_self=False,
                                    allow_forward=True,
                                    allow_backward=False,
                                    frames_up=0)

    # Define the macros
    jmp1 = ugp.Macro("    {jmp_instr} {jmp_ref}", {'jmp_instr': jmp_instructions,
↪'jmp_ref': jmp_target})
    instr_op_macro = ugp.Macro("    {instr} {regS}, {regD}",{'instr': instr_param,
↪'regS': reg_param, 'regD': reg_param})
    shift_op_macro = ugp.Macro("    {shift} ${int8}, {regD}", {'shift': shift_param,
↪'int8': int8, 'regD': reg_param})
    branch_macro = ugp.Macro("{branch} {jmp}", {'branch': jmp_instructions, 'jmp':␣
↪jmp_target})
    prologue_macro = ugp.Macro('    .file   "solution.c"\n' +
                               '    .text\n' +
                               '    .globl  _darwin\n' +
                               '    .def    _darwin;        .scl    2;      .type   ␣
↪32;     .endef\n' +
                               '_darwin:\n' +
```

(continues on next page)

(continued from previous page)

```
                                'LFB17:\n' +
                                '    .cfi_startproc\n' +
                                '    pushl   %ebp\n' +
                                '    .cfi_def_cfa_offset 8\n' +
                                '    .cfi_offset 5, -8\n' +
                                '    movl    %esp, %ebp\n' +
                                '    .cfi_def_cfa_register 5\n')
    init_macro = ugp.Macro("    movl ${int_a}, %eax\n" +
                           "    movl ${int_b}, %ebx\n" +
                           "    movl ${int_c}, %ecx\n" +
                           "    movl ${int_d}, %edx\n",
                           {'int_a': integer, 'int_b': integer, 'int_c': integer,
↪'int_d': integer})
    epilogue_macro = ugp.Macro(
        '    movl    %eax, -4(%ebp)\n' +
        '    movl    -4(%ebp), %eax\n' +
        '    leave\n' +
        '    .cfi_restore 5\n' +
        '    .cfi_def_cfa 4, 4\n' +
        '    ret\n' +
        '    .cfi_endproc\n' +
        'LFE17:\n' +
        '   .ident  "GCC: (MinGW.org GCC-8.2.0-5) 8.2.0"\n')

    # Define section
    sec1 = ugp.make_section({jmp1, instr_op_macro, shift_op_macro}, size=(1, 50))

    # Create a constraints library
    library = ugp.Constraints(file_name="solution{node_id}.s")
    library['main'] = [prologue_macro, init_macro, sec1, epilogue_macro]

    # Define the evaluator script and the fitness type
    if sys.platform != "win32":
        exit(-1)
    else:
        script = "eval.bat"
    library.evaluator = ugp.fitness.make_evaluator(evaluator=script, fitness_type=ugp.
↪fitness.Lexicographic)

    # Create a list of operators with their arity
    operators = ugp.Operators()
    # Add initialization operators
    operators += ugp.GenOperator(ugp.create_random_individual, 0)
    # Add mutation operators
    operators += ugp.GenOperator(ugp.hierarchical_mutation, 1)
    operators += ugp.GenOperator(ugp.flat_mutation, 1)
    operators += ugp.GenOperator(ugp.add_node_mutation, 1)
    operators += ugp.GenOperator(ugp.remove_node_mutation, 1)
    # Add crossover operators
    operators += ugp.GenOperator(ugp.macro_pool_one_cut_point_crossover, 2)
    operators += ugp.GenOperator(ugp.macro_pool_uniform_crossover, 2)

    # Create the object that will manage the evolution
    mu = 10
    nu = 20
    sigma = 0.7
    lambda_ = 7
```

(continues on next page)

(continued from previous page)

```
    max_age = 10

    darwin = ugp.Darwin(
        constraints=library,
        operators=operators,
        mu=mu,
        nu=nu,
        lambda_=lambda_,
        sigma=sigma,
        max_age=max_age,
    )

    # Evolve
    darwin.evolve()

    # Print best individuals
    logging.bare("These are the best ever individuals:")
    best_individuals = darwin.archive.individuals
    ugp.print_individual(best_individuals, plot=True, score=True)

    ugp.logging.cpu_info("Program completed")
    sys.exit(0)
```

The script syntax has been built to work with Windows 10, 64-bit, for *GCC: (MinGW.org GCC-8.2.0-5) 8.2.0*.

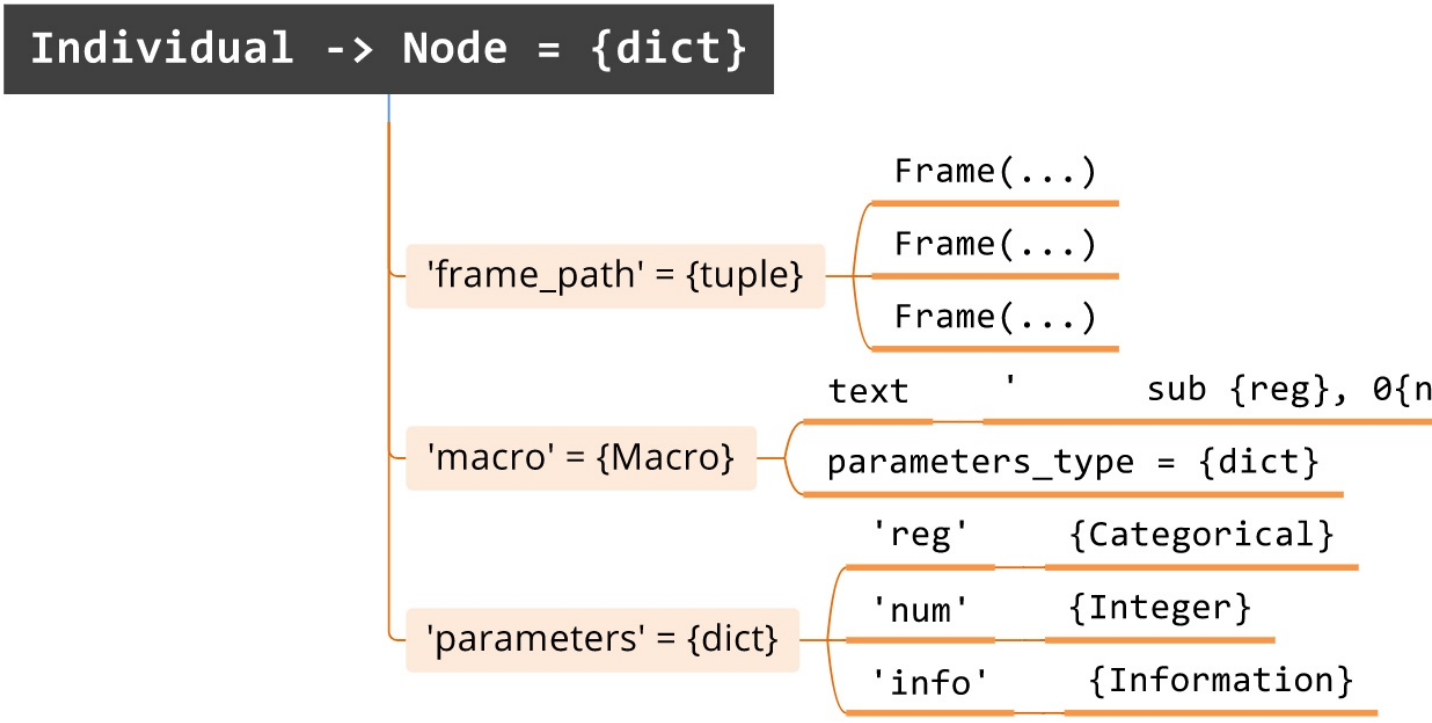
CHAPTER 4

Individual

The *microgp.individual.Individual* object represents a possible solution, the `microgp.individual.Individual._graph` attribute is a NetworkX MultiDiGraph, the nodes, inside it, are identified by their unique node_id (*microgp.node.NodeID* object that inherits from *int*).

Each *node* contains:

- a *microgp.macro.Macro* object;
- a dictionary that contains `microgp.parameter.Parameter` as values;
- a tuple of *microgp.common_data_structures.Frame*.



4.1 Individual

microgp.individual

class microgp.individual.**Individual**(*constraints: microgp.constraints.Constraints*, *graph: networkx.classes.multidigraph.MultiDiGraph = None*)

A solution encoded in MicroGP, the unit of selection in the evolutive process. Individuals are directed multigraph (*MultiDiGraph* in NetworkX), that is, more than one directed edges may connect the same pair of nodes. An individual must be finalized to be printed and copied, once it is finalized (individual._finalized == True) it can't be modified.

What an Individual contains:

- *node_id* (int): is the unique node_id of the individual
- *_constraints* (Constraints): is a reference to the constraints used to generate them
- *_graph* contains a reference to the nx MultiDiGraph. Inside it, node are identified by their unique node_id (*NodeID* object) and their data are stored as attributes:
  - *macro*: a reference to the macro
  - *parameters*: a dictionary with the actual parameters values
  - *frame_path*: path of the node in the constraints hierarchy
- *_operator* (callable): contains the pointer to the operator with which it was created
- *_parents* (Set[Individual] or None): set of parent individuals from which the individual was created. None if it has no parents
- *_age* (int): count of how many epochs the individual lived
- *_canonic_phenotype* (str): phenotype of the individual. It can be retrieved by printing the individual. It must be finalized

Parameters

- **constraints** – constraints of the individual
- **copy_from** – Individual to clone (if specified)

Examples:

- Create a new individual with its constraints

```
>>> first_individual = Individual(constraints=constraints)
```

- Clone an individual (same graph_manager, same parameter values, different NodeIDs and node_id)

```
>>> copied_individual = Individual(constraints=first_individual.constraints, copy_
↪from=first_individual)
```

- Print the phenotype representation of an individual

```
>>> print(first_individual)
```

add_node(*parent_node: Optional[microgp.node.NodeID], macro: microgp.macro.Macro, frame_path: Sequence[microgp.common_data_structures.Frame]*) → microgp.node.NodeID

Adds a node in the individual and chains it

Parameters

- **parent_node** (*NodeID or None*) – the previous block in the dump, if
- **the node will be the root of the individual** (*None*) –
- **macro** (*Macro*) – the macro that the node will contain
- **frame_path** (*Sequence of Frame*) – the full frame path to the section containing the node.

Returns The NodeID of the node just created

copy_parameters(*source_individual: microgp.individual.Individual, node_translation: Dict[microgp.node.NodeID, microgp.node.NodeID], destination_node_id: microgp.node.NodeID*)

Create parameters of the given node and copy their values from the original node (=‘‘node_translation.keys()‘‘)

Parameters

- **source_individual** (*Individual*) – individual that contains the parameter values of the nodes to copy
- **node_translation** (*Dict[NodeID, NodeID]*) – correspondences between source node ids and destination node ids
- **destination_node_id** (*NodeID*) – NodeID of the node to copy

Returns Dictionary that contains the parameters of the new copied node

copy_section_node_structure(*source_individual: microgp.individual.Individual, source_head: microgp.node.NodeID = None*) → Dict[microgp.node.NodeID, microgp.node.NodeID]

Copy the structure of a section from the *source_individual* to *self*. Used by the builder of the Individual when *copy_from* is not *None*.

Parameters

- **source_individual** (*Individual*) – Individual from which to copy the section structure
- **source_head** (*NodeID*) – NodeID of the head of the *next-chain* (section)

Returns The correspondences between source ‘NodeID‘s and destination ‘NodeID‘s

draw(**args*, *edge_color=None*, *with_labels=True*, *node_size=700*, *node_color=None*, ***kwargs*)

Draws the individual. All parameters are passed to nx.draw. If *node_color == None* then each *next-chain* will be colored with different colors (max 10 colors).

finalize() → None

Final setup of an individual: manage the pending movable nodes, remove non-visitable nodes from the graph_manager, initialize the frame tree and the properties, set the canonical representation of an individual. :return:

frames(*section: microgp.constraints.Section = None, section_name: str = None*) → Set[microgp.common_data_structures.Frame]

Get all frames of an individual belonging to a given section

Parameters

- **section** (*Section*) – limit to frames belonging to the section
- **section_name** (*str*) – limit to frames belonging to the section (name)

Returns A set of frames

get_next (*node: microgp.node.NodeID*) → microgp.node.NodeID
Get the successor of node (ie. the next block in the dump)

Parameters **node** (NodeID) – NodeID of the node of which I need the next

Returns NodeID of the next node, None if it is the last one in the graph_manager

get_predecessors (*node: microgp.node.NodeID*) → List[microgp.node.NodeID]
Get the list of al predecessors of a given node (following next's)

Parameters **node** (NodeID) – NodeID of the node of which I need the predecessors

Returns The list of NodeID of preceding nodes, None if there are none

get_unique_frame_name (*section: Union[microgp.constraints.Section, str]*) → str
Get a name never used in the individual by any other frame

Parameters **section** (Section) – section to which the frame refers (Section object or name of the section)

Returns The unique name of the new frame

link_movable_nodes () → Optional[bool]
Set the value of the parameters of type LocalReference and ExternalReference when the node is movable

randomize_macros () → None
Randomize the values of the parameters inside the macros

remove_node (*node_to_delete: microgp.node.NodeID*) → None
Delete the node from the graph_manager and connect the edges of the currently connected nodes

Parameters **node_to_delete** (NodeID) – Node to delete

run_paranoia_checks () → bool
Checks the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns *True*, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when *-O* is used for compiling). Their typical usage is: *assert foo.run_paranoia_checks()*

Returns True (always)

Raise: AssertionError if some internal data structure is incoherent

set_canonical () → None
Set the canonical representation of an individual (*self._canonic_phenotype*)

stringify_node (*node: microgp.node.NodeID*) → str
Generates the string with the node's current parameters

Parameters **node** – NodeID of the node to be converted into string

Returns The string that describes the macro and its parameters of the selected node

valid
Test whole set of *checkers* to validate the individual

Returns True if the individual have passed all the tests, False otherwise

4.1.1 NodesCollection

class microgp.individual.**NodesCollection**(*individual: Individual*, *nx_graph: networkx.classes.multidigraph.MultiDiGraph*)

Collection of nodes of an individual; quite but not completely different from a NetworkX node view.

When used as a dictionary it allows read-only access to NodeViews. E.g.,

```
>>> for n in ind.node_list;
>>>     print(ind.node_list[n]['foo'])
>>>     ind.node_list[n]['bar'] = 41    # nodes can be modified as a dictionary
>>>     ind.node_list[n].baz += 1       # or as properties
```

When *NodesCollection* is used as a function it allows to select nodes using various filters, e.g.,

```
>>> ind.node_list(section_selector='main', heads_selector=False, data=True)
```

Parameters

- **data** – When data is `None`, the return value is a list of *microgp.node.NodeID()*. When data is `True`, the return value is dictionary of dictionaries `{node_id: {<all node properties>}}`. When data is the key of a node property, the return value is a dictionary `{node_id: <the specified field>}`
- **default** – When property selected by data does not exists, the node is included in the result withe the specified value. If `default` is `None`, the node is not included in the result.
- **select_section** (*str or Section*) – Only include nodes belonging to the specified section.
- **select_frame** (*str or Frame*) – Only include nodes belonging to the specified frame.
- **select_heads** (*None or bool*) – if specified, return only nodes that are heads of sections (`True`); or nodes that are internal to sections (`False`)

Returns Either a list or a dictionary, see *data*

4.1.2 GraphWrapper_DELETED

4.1.3 Graph information getter methods

individual.**get_nodes_in_frame**(*frame: microgp.common_data_structures.Frame*, *frame_path_limit: int = None*) → List[microgp.node.NodeID]

Gets all nodes of an individual inside a given frame

Parameters

- **individual** (`Individual`) – the individual
- **frame** (`Frame`) – the frame
- **frame_path_limit** (*int*) – how deep is the path for matching (positive: from root, negative: from leaf)

Returns A list of of Nodes

individual.**get_nodes_in_section**(*section: microgp.constraints.Section*, *frame_path_limit: int = None*, *head: bool = False*) → List[microgp.node.NodeID]

Gets all nodes of an individual inside a given frame

If *frame_path_limit* is set to N with N > 0, only the first N frames are considered for the match. If With N < 0, only the last N frames are considered for the match.

Parameters

- **individual** (Individual) – the individual
- **section** (Section) – the section
- **frame_path_limit** (*int*) – limit the frame path
- **head** (*bool*) – returns only the head of the section

Returns The list of nodes in the selected section

individual.**get_frames**(*section: microgp.constraints.Section = None*, *section_name: str = None*) → Set[microgp.common_data_structures.Frame]

Gets all frames of an individuals belonging to a given section

Parameters

- **individual** (Individual) – the individual
- **section** (Section) – limit to frames belonging to the section
- **section_name** (*str*) – limit to frames belonging to the section (name)

Returns A set of frames

individual.**get_macro_pool_nodes_count**(*frames: Set[microgp.common_data_structures.Frame] = None*) → Dict[microgp.common_data_structures.Frame, int]

Get a dict containing {Frame: number_of_macros}. Selects only MacroPools

Parameters

- **frames** (*Set[Frame]*) – set of frames of which I want the number of nodes
- **individual** (Frame) – individual from which count the nodes

Returns Dictionary containing the amount of nodes (value) for each Frame (key)

Meta private

individual.**check_individual_validity**() → bool

Check an individual against its constraints.

Check the validity of all parameters (e.g., range), then the default *Properties* (e.g., number of macros in sections), and finally all the custom *Properties* added by the user.

The result is cached, as individuals must be *finalized* to be checked

Parameters **individual** (Individual) – the individual to be checked

Returns The validity as a boolean value

4.2 Macro

microgp.macro

class microgp.macro.**Macro**(*text: str*, *parameters_type: dict = None*)

The blueprint of macro.

A "macro" is a fragment of text with zero or more variable parameters. it is the building block of a solution. A macro is associated with the node in the DAG encoding the individual.

Notes

Attributes are read-only and can only be set when the macro is created. Some parameters are by default available to all macros, the list is in Macro.MAGIC_PARAMETERS

add_parameter(*name: str*, *parameter_type: microgp.parameter.abstract.Parameter*) → None
Add parameters to the macro

Parameters

- **name** – name of the parameter
- **parameter_type** – type of the parameter

run_paranoia_checks() → bool
Checks the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns *True*, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when *-O* is used for compiling). Their typical usage is: *assert foo.run_paranoia_checks()*

Returns True (always)

Raise: AssertionError if some internal data structure is incoherent

4.3 Node

microgp.node

class microgp.node.**NodeID**(*value: Optional[int] = None*)
A node in the directed MultiDiGraph describing the individual.

Use *n = NodeID()* to get a unique node_id.

Use *n = NodeID(int)* to get a specific node node_id. This is deprecated as it could only be useful during debug.

is_valid(*value: Any*) → bool
Checks an object against a specification. The function may be used to check a value against a parameter definition, a node against a section definition).

Returns True if the object is valid, False otherwise

4.4 Frame

A Frame is a unique instance of a section, its name is unique and can be generated by the *microgp.individual.Individual.get_unique_frame_name* method in the following way:

```
>>> Frame(individual.get_unique_frame_name(section), section)
```

class microgp.common_data_structures.**Frame**(*name*, *section*)

name
Alias for field number 0

section
Alias for field number 1

CHAPTER 5

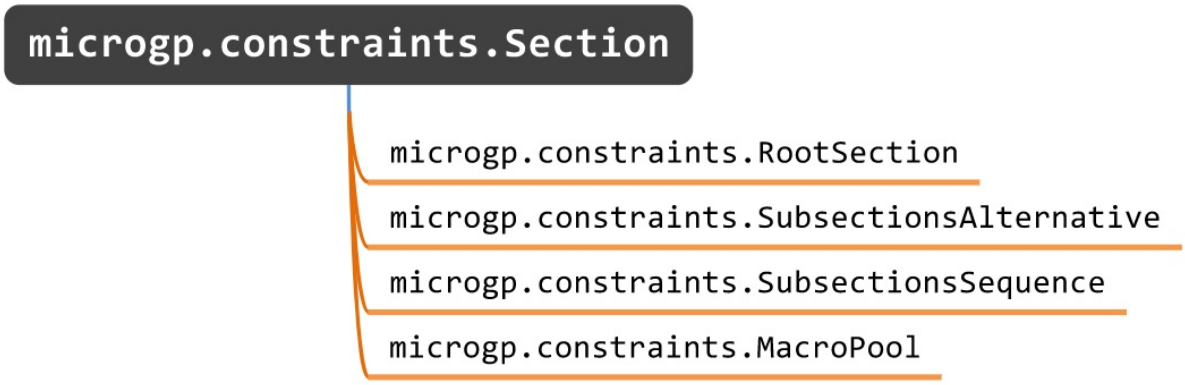
Constraints

The `microgp.constraints.Constraints` class contains the set of macros and sections of which an individual is composed, the *`microgp.properties.Properties`* and the function that will evaluate the individuals (`microgp.contraints.Constraints._evaluator`).

5.1 Section

A section can be of several types:

- a *`microgp.constraints.MacroPool`* (pool of macros)
- a *`microgp.constraints.RootSection`* (root section of an individual)
- a *`microgp.constraints.SubsectionsSequence`* (sequence of sections)
- a *`microgp.constraints.SubsectionsAlternative`* (sequence of sections that can be alternatively chosen)



class microgp.constraints.**Section**(*name: str*, *instances: Optional[Tuple[int, int]] = None*, *label_format=None*)

Base structural unit. A section can be composed by one or more macros, a set or a list of other subsections. See

microgp.constraints.make_section.

5.1.1 How a section is built

microgp.constraints.make_section

To build a section the following method is used:

constraints.**make_section**(*name: str = None*, *instances: Optional[Tuple[int, int]] = None*, *size: Tuple[int, int] = None*, *label_format: str = None*) → microgp.constraints.Section

Builds a section from a human-readable description.

Parameters

- **section_definition** – macro, list of macros or set of macros that will be translated into MacroPool / SubsectionsSequence / SubsectionsAlternative
- **name** (*str*) – Name of the section to build
- **instances** (*tuple(int,int)*) – (None or (int >=0, int >0)) How many times the section can appear inside an individual
- **size** (*tuple(int,int)*) – ((int >=0, int >0)) number of macro that the section can contain
- **label_format** (*str*) – define how to translate a node into string

Returns The section just built

Examples:

- Create a section of name *word_sec* containing a macro (word_macro), it will appear once inside the individual

```
>>> word_section = ugp4.make_section(word_macro, size=(1, 1), name='word_sec')
```

- Create a section of name *sec_jmp* that contains 1 to 4 macros (jmp1), it will appear once inside the individual

```
>>> sec_jmp = ugp4.make_section(jmp1, size=(1, 4), name='sec_jmp')
```

- Create a section with a default unique name that contains 2 to 5 macros chosen in {add, sub} and it can appear 0 to 10 times inside the individual

```
>>> generic_math = ugp4.make_section({add, sub}, size=(2, 5), instances=(0,␣
↪10))
```

- Build the **main** section with 3 sections, the second one is a *SubsectionsSequence* that contains 3 sections:
  - A *SubsectionsAlternative* {sec2a, sec2b}
  - A simple section sec_jmp (*MacroPool*)
  - A simple section containing a Macro without parameters (*MacroPool*)

```
>>> library['main'] = [
>>>     'Prologue...'
>>>     [{sec2a, sec2b}, sec_jmp, '; this is a comment'],
>>>     'Epilogue...'
>>> ]
```

The **main** section is contained inside the *RootSection*.

5.1.2 RootSection

class microgp.constraints.**RootSection**
The ROOT section of an individual. Each individual have one and only one root section.

5.1.3 MacroPool

class microgp.constraints.**MacroPool**(*macro_pool: Collection[microgp.macro.Macro] = None*, *name: str = None*, *size: Tuple[int, int] = (1, 1)*, ***kwargs*)
A pool of macros.

5.1.4 SubsectionsAlternative

class microgp.constraints.**SubsectionsAlternative**(*name: str*, *sub_sections: Sequence[microgp.constraints.Section] = None*, ***kwargs*)
A list of alternative (sub)sections. This type of section contains a sequence of sections.

5.1.5 SubsectionsSequence

class microgp.constraints.**SubsectionsSequence**(*sub_sections: Sequence[microgp.constraints.Section] = None*, *name: str = None*, ***kwargs*)
A sequence of subsections. This class contains a tuple of sub sections

5.2 Properties

Properties are boxes that can contain values and checkers that run tests on the values. The testers can return True or False. Values in Properties can be customized, for instance a value can be the number of macros in a certain section and can be set a checker on it that checks that this values doesn't exceed a certain threshold.

Builders can be:

- *custom_builders*: customizable by the user;
- *default_builders*: builders provided with MicroGP package.

Another distinction:

- *base_builders*: I can set a certain value;
- *cumulative_builders*: I can set a value and it can be added up recursively going through the frame tree.

This checkers are called by *microgp.individual.check_individual_validity*

microgp.properties

class microgp.properties.**Properties**
Updates a dictionary of *values* and runs *checks* against them.

Properties are used to check if a frame (ie. the portion of the individual implementing a given section) is valid. First, all functions registered as *values builders* are called, then all functions registered as *check* are evaluated; if all succeeded, then True is turned.

Values are divided in *custom* and *base*. User's builders build custom ones. Values can be retrieved through property *values* that merge the two, alternatively they can be retrieved through properties *base_values* and *custom_values*.

Values builders are functions returning a dictionary of values *{'value_name': value}* that is added to the current value-bag. Values cannot be shadowed.

Checks are called when the value bag is complete and get getting all values as parameters, i.e. *check(**values)*

Examples: create two cumulative (custom) builders and add a checker that test that two sections have the same number of nodes

```
>>> sec2a.properties.add_cumulative_builder(lambda num_nodes, **v: {'sec2a': num_
↪nodes})
>>> sec2b.properties.add_cumulative_builder(lambda **v: {'sec2b': v['num_nodes']})
>>> library.global_properties.add_check(lambda sec2a, sec2b, **v: sec2a == sec2b)
```

base_values
Read-only dictionary of base values

cumulative_values
Read-only dictionary of custom values

run_checks()
Runs all checks against current values; returns True if all succeed.

update_values(*cumulative_values: Optional[dict] = None*, ***kwargs*)
Runs properties builders and collect the results into *values* dictionary.

values
Read-only dictionary of all values (both custom and base)

5.2.1 Default builders

microgp.default_builders

microgp.default_builders.**default_base_builder**(*individual: Individual*, *frame: microgp.common_data_structures.Frame*, ***kwargs*) → Dict[str, Any]

Get base stats of the frame

microgp.default_builders.**default_cumulative_builder**(*individual: Individual*, *frame: microgp.common_data_structures.Frame*, ***kwargs*) → Dict[str, Any]

Get base cumulative stats of the frame

CHAPTER 6

Darwin

The *microgp.darwin.Darwin* class is in charge of handling the whole evolutionprocess. It contains:

- *Constraints* (microgp.constraints.Constraints) that must be the same for each individual managed by Darwin;
- *Population* (*microgp.population.Population*) that manages the set of individuals present in the current generation;
- *Archive* (*microgp.archive.Archive*) that manages the best individuals ever contained in the population;
- *Operators* (*microgp.operators.Operators*) that wraps all the *GenOperator* (*microgp.genoperator.GenOperator*), manages statistics, and operator selection;
- other evolution parameters such as selection pressure, population size, number of operators to use for the first generation, maximum age of an individual, etc;

microgp.darwin

class microgp.darwin.**Darwin**(*constraints: microgp.constraints.Constraints*, *operators: microgp.operators.Operators*, *mu: int*, *lambda_: int*, *nu: int = None*, *tau: float = 2.0*, *strength: float = 0.5*, *max_age: Optional[int] = None*, *max_generations: Optional[int] = 42*, *stopping_conditions: Optional[list] = None*)

This class manages the evolution, stores the genetic operators, the population, and the archive. You can set some evolution parameters (*lambda, tau, nu, strength, mu*) and a list of stopping conditions. A *microgp.population.Population* and a *microgp.population.Archive* objects are also initialized.

Parameters

- **constraints** (*Constraints*) – Constraints object that each individual managed by Darwin must have.
- **operators** (Operators) – operators object that contains the set of GenOperators that will manipulate the individuals.
- **mu** (*int*) – Population size.

- **lambda** (*int*) – Number of operators to pick at each generation (except when the population is empty).
- **nu** (*int*) – Number of initialization operators to pick when the population is empty. None if you want mu individuals.
- **tau** (*float*) – Selection pressure. Default value: 2.0
- **strength** (*float*) – Probability and strength of the mutation (to be passed to the mutation GenOperators).
- **max_age** (*int*) – Maximum age that an individual in the population can have. None if you don't want to filter individuals by age.
- **stopping_conditions** –

Examples:

- Initialize a *Darwin* object:

```
>>> mu = 10
>>> nu = 20
>>> strength = 0.2
>>> lambda_ = 7
>>> max_age = 10
>>> darwin = ugp4.Darwin(
>>>     constraints=library,
>>>     operators=operators,
>>>     mu=mu,
>>>     nu=nu,
>>>     lambda_=lambda_,
>>>     strength=strength,
>>>     max_age=max_age)
```

- Evolve, print results (*Population*):

```
>>> darwin.evolve()
>>> logging.bare("This is the population:")
>>> for individual in darwin.population:
>>>     msg = 'Printing individual ' + individual.id
>>>     ugp4.print_individual(individual, msg=msg, plot=True)
>>>     ugp4.logging.bare(individual.fitness)
```

- Print the *Archive* that contains the best eve individuals

```
>>> logging.bare("These are the best ever individuals:")
>>> ugp4.print_individual(darwin.archive)
```

do_generation() → None
Perform a generation of the evolution. Pick lambda (or nu) operators, clean the resulting set of individuals given by the operators, join it to population and keep the best mu individuals

evolve() → None
Evolve the population until at least one of the stopping conditions becomes True

filter_offspring(*temporary_offspring: Optional[List[Optional[microgp.individual.Individual]]]*) → Optional[List[Optional[microgp.individual.Individual]]]
Remove "None" elements and choose the best element in sublist recursively

Parameters **temporary_offspring** – the list of individuals to just generated by the operator

Returns List of valid individuals

get_best_unpacking(*individuals: list*) → Optional[microgp.individual.Individual]
Find the best value in the given list (recursive)

keep_at_most_mu() → None
Keep in the population at most mu individuals removing the worst

update_archive() → None
Insert in archive the best individuals (and remove the no more good enough individuals)

6.1 Population

microgp.population

class microgp.population.**Population**
This class contains the set of individuals composing the population and manages the selection, insertion, removal of the individuals based on the age or whether their phenotype is already in the population. A non-valid individual can't be inserted.

Examples:

- Add a set of individuals in the population

```
>>> darwin.population += set(list_of_individuals)
```

- Add a single individual in the population

```
>>> darwin.population += set(individual_A)
>>> darwin.population += set(individual_B)
```

- Remove a single individual from the population

```
>>> darwin.population -= set(individual_A)
```

- Remove multiple individuals (set) from the population

```
>>> darwin.population = darwin.population - set(individual_A)
```

- Retrieve from population an individual using tournament selection Tournament selection

```
>>> selected_individual = darwin.tournament_selection(tau)
```

- Retrieve the entire set of individual contained in the population

```
>>> population = darwin.population.individuals
```

filter_by_age(*max_age: int = None*) → None
Remove from the population the individuals that are too old (individual.age >= max_age)

static filter_clones() → None
Check whether or not there are individuals with the same (canonical) phenotype and then remove them

static grow_old(*individuals: Set[microgp.individual.Individual]*) → None
Increment the age of a set of individuals. Typically the set is {Population - Archive}

select(*tau: float = 2*) → microgp.individual.Individual
Select an individual from the population calling the tournament_selection(tau) method

tournament_selection(*tau: float = 2*) → microgp.individual.Individual
Run several tournaments among a few (floor(tau) or ceil(tau)) individuals and return the best one based on the fitness

6.2 Archive

`microgp.archive`

class microgp.archive.**Archive**
This class manages the set of individuals not dominated by all other individuals currently or previously contained in the `microgp.darwin.Darwin._population`.

Examples:

- Try to insert an individual in the archive:

```
>>> self._archive += individual
```

The individual will be inserted only if it is not dominated by all individual already inside the archive. If it is not dominated then the individuals that just became dominated are removed from the archive.

6.3 Operators

microgp.operators

class microgp.operators.**Operators**
This class wraps all operators, manages statistics, and GenOperator selection. The selection is made on the basis of the arity and the success and failure statistics of the operator.

Examples:

- Create and fill an Operators object to be passed to a Darwin object

```
>>> operators = ugp4.Operators()
>>> init_op1 = ugp4.GenOperator(ugp4.create_random_individual, 0)
>>> operators += init_op1
>>> mutation_op1 = ugp4.GenOperator(ugp4.remove_node_mutation, 1)
>>> operators += mutation_op1
>>> crossover_op1 = ugp4.GenOperator(ugp4.switch_proc_crossover, 2)
>>> operators += crossover_op1
>>> crossover_op2 = ugp4.GenOperator(ugp4.five_individuals_crossover, 5)
>>> operators += crossover_op2
```

- Select k operators that has arity in the given range

```
>>> selected_operators = operators.select(max_arity=0, k=10)
>>> selected_operators = operators.select(min_arity=1, max_arity=2 k=20)
```

select (*min_arity: int = 0*, *max_arity: int = None*) → microgp.genoperator.GenOperator
Select a set of operators with arity in [min_arity, max_arity]

Parameters

- **min_arity** (*int*) – minimum arity of the operators that will be returned
- **max_arity** (*int*) – maximum arity of the operators that will be returned
- **k** (*int*) – number of genetic operators to return

Returns a valid genetic operator

6.3.1 GenOperator

microgp.genoperator

class microgp.genoperator.**GenOperator** (*function: collections.abc.Callable*, *arity: int*)
Wrapper of a method that implements the algorithm manipulating or building one or more individuals. This class will also manage (in the future versions) the statistics applied to the assigned method. The method wrapped in the GenOperator must have **kwargs in its parameters.

Examples:

- Build three genetic operators passing the method and the arity

```
>>> init_op1 = ugp4.GenOperator(ugp4.create_random_individual, 0)
>>> mutation_op1 = ugp4.GenOperator(ugp4.remove_node_mutation, 1)
>>> crossover_op1 = ugp4.GenOperator(ugp4.switch_proc_crossover, 2)
>>> crossover_op2 = ugp4.GenOperator(ugp4.five_individuals_crossover, 5)
```

- Call the method inside the genetic operator

```
>>> selected_crossover_genoperator(individual1, individual2)
>>> selected_mutation_genoperator(individual, strength=0.7,␣
↪constraints=constraints))
>>> individuals = tuple(ind1, ind2, ind3, ind4, ind5)
>>> kwargs = {'param1': var1, 'param2': var2, 'param3': [a, b, c, d, e]}
>>> selected_crossover_5_individuals(*individuals, kwargs)
```

CHAPTER 7

Individual Operators

MicroGP4 package offers some basic operators that can be found in `microgp.individual_operators`. Three of them are crossover operators, four are mutation operators and one has the goal to create a random individual. As you can see in *Darwin*, the methods that are listed in this module can be passed to the constructor of a *GenOperator* an then added to the list of operators used by the *`microgp.darwin.Darwin`* object.

`microgp.individual_operators`

7.1 Initialization operator

This kind of operator doesn't receive any individual as input and returns a new individual.

7.1.1 Create random individual

7.2 Mutation operators

This kind of operator change one or more genes that describe an individual. The intensity of the change and the probability that it takes place depend on **sigma**. This is a parameter specified during the creation of the *Darwin* object and it can assume values in [0, 1].

The available mutation operators are:

- `microgp.individual_operators.remove_node_mutation`
- `microgp.individual_operators.add_node_mutation`
- `microgp.individual_operators.hierarchical_mutation`
- `microgp.individual_operators.flat_mutation`

7.2.1 Remove node mutation

7.2.2 Add node mutation

7.2.3 Hierarchical mutation

7.2.4 Flat mutation

7.3 Crossover operators

The available crossover operators are:

- Switch procedure crossover: `microgp.individual_operators.switch_proc_crossover`
- MacroPool OneCut crossover: `microgp.individual_operators.macro_pool_one_cut_point_crossover`
- MacroPool Uniform crossover: `microgp.individual_operators.macro_pool_uniform_crossover`

7.3.1 Switch procedure crossover

7.3.2 MacroPool OneCut crossover

7.3.3 MacroPool Uniform crossover

CHAPTER 8

Fitness

The library allows the use two main types of fitnesses (FitnessTuple, FitnessTupleMultiobj), both inherit from the Base class.

8.1 Base fitness class

microgp.fitness.base

class microgp.fitness.base.**Base**

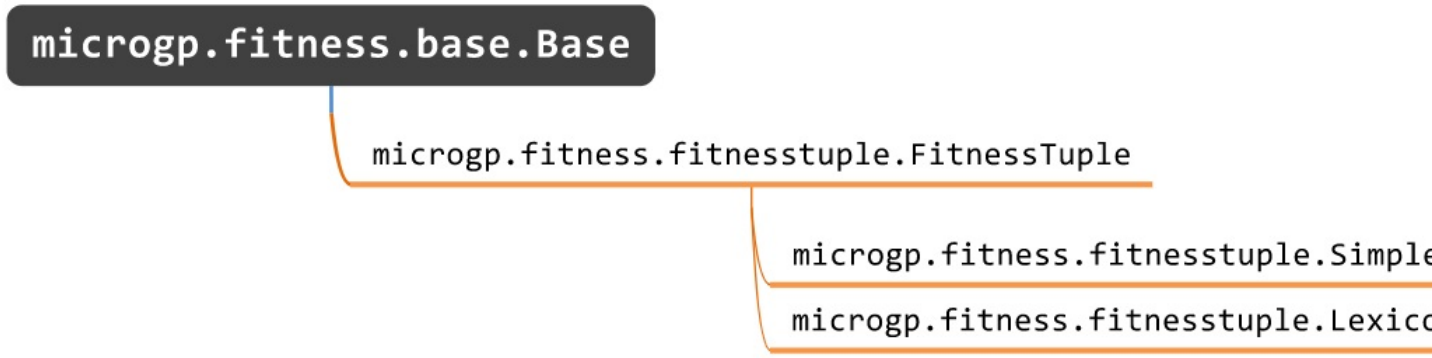
Base class for storing fitness.

The different selection schemes are implemented simply by overriding the > (greater than) operator. See the other fitness.Classes for details.

Please note that, according to Spencer's 'Survival of the Fittest', the bigger the better. Thus we are *maximizing* the fitness and not minimizing a mathematical function.

The method calling the correspondent `Base.sort()` is:

8.2 Fitness Tuple



microgp.fitness.fitnesstuple

class microgp.fitness.fitnesstuple.**FitnessTuple**
Fitness used for single objective purposes.

static sort (*fmap: List[Tuple[Any, Type[FitnessTuple]]]*) → List[Tuple[Any, Type[microgp.fitness.fitnesstuple.FitnessTuple]]]
Sort a list of tuple (Any, FitnessTuple)

Parameters **fmap** – list of tuple (Any, FitnessTuple)

Returns an ordered list of tuples (Any, FitnessTuple)

microgp.fitness.simple

class microgp.fitness.simple.**Simple**
The simplest possible fitness: a single value

microgp.fitness.lexicographic

class microgp.fitness.lexicographic.**Lexicographic**
Tuples able to smoothly handle single- and multi-value fitnesses

8.3 Fitness Tuple Multiobjective

microgp.fitness.fitnesstuplemultiobj

class microgp.fitness.fitnesstuplemultiobj.**FitnessTupleMultiobj**
Fitness used for multiple objective purposes.

static sort (*fmap: List[Tuple[Any, Type[FitnessTupleMultiobj]]]*) → List[Tuple[Any, Type[microgp.fitness.fitnesstuplemultiobj.FitnessTupleMultiobj]]]
Sort a list of tuple (Any, FitnessTupleMultiobj) using Pareto front

Parameters **fmap** – list of tuple (Any, FitnessTupleMultiobj)

Returns an ordered list of tuples (Any, FitnessTupleMultiobj)

Sorting example considering an input of type List[Individual, FitnessTupleMultiobj].

```
real_order[
    {ind3, ind2},
    {ind5, ind1, ind7},
    {ind4},
    {ind6, ind8},
    {ind9}
]
ranking = {
    (ind1, ind1.fitness): 2,
    (ind2, ind2.fitness): 1,
    (ind3, ind3.fitness): 1,
    (ind4, ind4.fitness): 3,
    (ind5, ind5.fitness): 2,
    (ind6, ind6.fitness): 4,
    (ind7, ind7.fitness): 2,
    (ind8, ind8.fitness): 4,
    (ind9, ind9.fitness): 5
}
returned_value = [
    (ind2, ind2.fitness)
    (ind3, ind3.fitness)
    (ind1, ind1.fitness)
    (ind5, ind5.fitness)
    (ind7, ind7.fitness)
    (ind4, ind4.fitness)
    (ind6, ind6.fitness)
    (ind8, ind8.fitness)
    (ind9, ind9.fitness)
]
```

microgp.fitness.lexicase

class microgp.fitness.lexicase.**Lexicase**
A fitness for using Lexicase Selection.

Lexicase Selection is a technique supposedly able to handle multi-objective problems where solutions must perform optimally on each of many test cases.

See 'Solving Uncompromising Problems With Lexicase Selection', by T. Helmuth, L. Spector, and J. Matheson <https://dx.doi.org/10.1109/TEVC.2014.2362729>

Note: as an alternative, consider using *Chromatic Selection*.

microgp.fitness.aggregate

class microgp.fitness.aggregate.**Aggregate**
Tuples able to smoothly handle single- and multi-value fitnesses

microgp.fitness.chromatic

class microgp.fitness.chromatic.**Chromatic**
A fitness for using Chromatic Selection.

Chromatic Selection is a fast, simple and grossly approximate technique for tackling multi-value optimization. See: 'Chromatic Selection – An Oversimplified Approach to Multi-objective Optimization', by G. Squillero <https://dx.doi.org/10.1007/978-3-319-16549-3_55>

Note: as an alternative, consider using *Lexicase Selection*.

8.4 Evaluator

microgp.fitness.evaluator

microgp.fitness.evaluator.**make_evaluator**(*evaluator: Union[str, callable], fitness_type: Type[microgp.fitness.fitnesstuple.FitnessTuple] = <class 'microgp.fitness.simple.Simple'>, num_elements: int = None*) → Callable

Build a fitness evaluator that calls a script.

Parameters

- **evaluator** (*str or callable*) – name of the script. The result is taken from stdout.
- **fitness_type** – kind of fitness, ie. type of comparison.
- **num_elements** (*int*) – number of relevant element in the script output, the remaining are considered "comment". If None, the all output is part of the fitness.

Returns A function that can be stored into Constraints.evaluator.

CHAPTER 9

Parameters

The parameters are used to make the macros dynamic. For example you can build a parameter of type *microgp.parameter.categorical.Categorical* passing some values (*alternatives*); the macro that contains this kind of parameter will take the given values and the resulting text of the macro will depend on the values taken by the parameters assigned. The method *microgp.parameter.helpers.make_parameter* allows to build a parameter providing the parameter class type and the needed attributes.

Examples:

```
>>> registers = ugp.make_parameter(ugp.parameter.Categorical, alternatives=['ax', 'bx
↪', 'cx', 'dx'])
>>> int256 = ugp.make_parameter(ugp.parameter.Integer, min=0, max=256)
>>> add = ugp.Macro("    add {reg}, 0{num:x}h ; ie. {reg} += {num}", {'reg':␣
↪registers, 'num': int256})
```

There are several types of parameters:

- *Integer* (*microgp.parameter.integer.Integer*);
- *Bitstring* (*microgp.parameter.bitstring.Bitstring*);
- *Categorical, CategoricalSorted* (*microgp.parameter.categorical*);
- *LocalReference, ExternalReference* (microgp.parameter.reference);
- *Information* (microgp.parameter.special.Information).

9.1 Parameter

Parameters inheritance hierarchies:

microgp.parameter.Parameter

microgp.parameter.Structural

microgp.parameter.Special

9.1.1 Integer

microgp.parameter.integer

class microgp.parameter.integer.**Integer**(*name: str*, **args*, ***kwargs*)
Integer parameter in a given range.

Example:

```
>>> int256 = ugp4.make_parameter(ugp4.parameter.Integer, min=0, max=256)
```

Parameters

- **min** (*int*) – minimum value **included**.
- **max** (*int*) – maximum value **not included**.

is_valid(*value*)
Check if the passed value is in range min, max.

mutate(*strength: float = 0.5*)
Mutate current value according to strength (ie. strength).

run_paranoia_checks() → bool
Checks the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns *True*, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when *-O* is used for compiling). Their typical usage is: *assert foo.run_paranoia_checks()*

Returns True (always)

Raise: AssertionError if some internal data structure is incoherent

9.1.2 Bitstring

microgp.parameter.bitstring

class microgp.parameter.bitstring.**Bitstring**(**args*, ***kwargs*)
Fixed-length bitstring parameter.

Example:

```
>>> word8 = ugp4.make_parameter(ugp4.parameter.Bitstring, len_=8)
```

Parameters **len_** (*int > 0*) – length of the bit string

is_valid(*value*)
Check whether the given value is valid for the parameters

mutate(*strength: float = 0.5*)
Mutate current value according to strength (ie. strength).

value
Get current value of the parameter (type str)

9.1.3 Categorical, CategoricalSorted

microgp.parameter.categorical

class microgp.parameter.categorical.**Categorical**(*name: str*, **args*, ***kwargs*)
Categorical parameter. It can take values in 'alternatives'.

Example:

```
>>> registers = ugp4.make_parameter(ugp4.parameter.Categorical, alternatives=['ax
↪', 'bx', 'cx', 'dx'])
```

Parameters **alternatives** (*list*) – list of possible values

is_valid(*value*)
Check whether the given value is valid for the parameters

mutate(*strength: float = 0.5*)
Mutate current value according to strength (ie. strength).

run_paranoia_checks() → bool
Checks the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns *True*, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when *-O* is used for compiling). Their typical usage is: *assert foo.run_paranoia_checks()*

Returns True (always)

Raise: AssertionError if some internal data structure is incoherent

class microgp.parameter.categorical.**CategoricalSorted**(*name: str*, **args*, ***kwargs*)
CategoricalSorted parameter. It can take values in 'alternatives'. It behaves differently during the mutation phase.

Example:

```
>>> cat_sor = ugp4.make_parameter(ugp4.parameter.CategoricalSorted, alternatives=[
↪'e', 'f', 'g', 'h', 'i', 'l'])
```

Parameters **alternatives** (*list*) – sorted list of possible values

mutate(*strength: float = 0.5*)
Mutate current value according to strength (ie. strength).

run_paranoia_checks() → bool
Checks the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns *True*, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when *-O* is used for compiling). Their typical usage is: *assert foo.run_paranoia_checks()*

Returns True (always)

Raise: AssertionError if some internal data structure is incoherent

9.1.4 LocalReference, ExternalReference

`microgp.parameter.reference`

class `microgp.parameter.reference.`**Reference**(*individual: Individual*, *node: NodeID*, ***kwargs*)

Base class for references. Inherits from Structural

class `microgp.parameter.reference.`**LocalReference**(*individual: Individual*, *node: NodeID*, ***kwargs*)

A reference to a Node connected through (undirected) "next" edges

Examples:

```
>>> ref_fwd = ugp4.make_parameter(ugp4.parameter.LocalReference,
>>>                               allow_self=False,
>>>                               allow_forward=True,
>>>                               allow_backward=False,
>>>                               frames_up=1)
>>> ref_bcw = ugp4.make_parameter(ugp4.parameter.LocalReference,
>>>                               allow_self=False,
>>>                               allow_forward=False,
>>>                               allow_backward=True,
>>>                               frames_up=1)
```

Parameters

- **allow_self** (*bool*) – The referenced node may be the node itself;
- **allow_forward** (*bool*) – The referenced node may be a successors;
- **allow_backward** (*bool*) – The referenced node may be a predecessors;
- **frames_up** (*int*) – How many frame up the reference must be within (optional, default: 0, i.e., only the current frame)

class `microgp.parameter.reference.`**ExternalReference**(*individual: Individual*, *node: NodeID*, ***kwargs*)

A reference to a NodeID non connected through (undirected) "next" edges.

```
>>> proc1 = ugp4.make_parameter(ugp4.parameter.ExternalReference, section_name=
↪'proc1', min=5, max=5)
```

Parameters **section_name** (*str*) – name of the new target section.

9.1.5 Information

`microgp.parameter.special.Information`

9.2 Helpers

microgp.parameter.helpers

microgp.parameter.helpers.**make_parameter**(*base_class: Type[microgp.parameter.abstract.Parameter]*, ***attributes*) → Type[CT_co]

Binds a Base parameter class, fixing some of its internal attributes.

Parameters

- **base_class** (*Parameter*) – Base class for binding parameter.
- ****attributes** (*dict*) – Attributes.

Returns Bound parameter.

Examples:

```
>>> register = ugp4.make_parameter(ugp4.parameter.Categorical, alternatives=['ax',
↪ 'bx', 'cx', 'dx'])
>>> int8 = ugp4.make_parameter(ugp4.parameter.Integer, min_=-128, max_=128)
>>> p1, p2 = register(), int8()
```

CHAPTER 10

Paranoid and Pedantic

The following classes inherit from *Paranoid* and/or *Pedantic*.

- **Pedantic**: *Parameter*, *Constraints*, *Individual*
- **Paranoid**: *Parameter*, *Constraints*, *Individual*, *Section*, *Macro*

10.1 Paranoid

microgp.abstract.Paranoid

class microgp.abstract.**Paranoid**
Abstract class: Paranoid classes do implement *run_paranoia_checks()*.

10.2 Pedantic

microgp.abstract.Pedantic

class microgp.abstract.**Pedantic**
Abstract class: Pedantic classes do implement *is_valid()*.

CHAPTER 11

Authors

11.1 Giovanni Squillero

Politecnico di Torino
Department of Control and Computer Engineering
Corso Duca degli Abruzzi 24
10129 Torino — Italy
E-mail: giovanni.squillero@polito.it

11.2 Alberto Tonda

Génie et Microbiologie des Procédés Alimentaires (GMPA)
French National Institute for Agricultural Research
AgroParisTech, Université Paris-Saclay
1 av. Brétignières
78850 Thiverval-Grignon — France
E-mail: alberto.tonda@inra.f

CHAPTER 12

License

A life spent making mistakes is not only more honorable,
but more useful than a life spent doing nothing.
— George Bernard Shaw (1856–1950)

CHAPTER 13

Acknowledgements

This Documentation was built with Sphinx using a theme provided by Read the Docs.

- modindex

Python Module Index

m

Index

T

U

V