
PyDGGA Documentation

Release 1.5.8 beta / 0.2.1

Logic Optimization Group

Mar 12, 2021

CONTENTS:

1	General Information	1
1.1	Background	1
1.2	Version	1
1.3	System Requirements	1
2	PyDGGA's User Manual	2
2.1	PyDGGA Command Line Tool	2
2.2	PyDGGA File Formats	10
2.3	PyDGGA Templates	15

GENERAL INFORMATION

1.1 Background

PyDGGA is a new distributed version of the Gender-Based Genetic Algorithm (GGA) for automatic algorithm configuration. GGA is an algorithm for finding the most appropriate settings of an algorithm for a given set of instances of a problem.

Although the core ideas are taken from GGA the actual implementation differs from that of the original paper, for more information about the original GGA please read the original paper: <https://doi.org/10.3233/978-1-60750-061-2-284>

1.2 Version

The current version of the **pydgga** package is 1.5.8 and the command line interface version is 0.2.1.

1.3 System Requirements

PyDGGA is distributed in a bundled binary format for each supported platform, which eliminates the problem of installing additional dependencies. Nonetheless, it is recommended to have some scripting language (Bash, Python, etc) to create wrapper programs for the target algorithm to translate the input/output from/to PyDGGA

PyDGGA has been designed to be inherently parallel and although it can work using a single CPU core, it is recommended to have at least 4 cores available. As for the memory requirements, this is affected by how PyDGGA is configured and by the target algorithm memory requirements. For PyDGGA alone, depending on how it is configured and if it runs locally or in a distributed environment we recommend having between 1 and 3 Gib of free memory.

PYDGGGA'S USER MANUAL

2.1 PyDGGGA Command Line Tool

In this section we will show you how to interact with the command line tool. The command line tool is able to run multiple sub-commands, this way the same tool can be used to run PyDGGGA locally, start the distributed PyDGGGA master or start a worker.

```
pydgga sub-command <sub-command arguments>
```

For simplicity, whenever we talk about pydgga (all in lowercase) we will refer to the command line tool. Also, for the sake of clarity this document will guide you via different examples for each sub-command, providing references to other commands usage sections when required. Finally, to make the document clearer, we assume that pydgga is installed in a location that is on your system's *PATH*.

2.1.1 Local Algorithm Configuration

There are two ways of running PyGGA using the *PyDGGGA Command Line Tool*. This section corresponds to the easiest one, running it locally in parallel using only one machine. In section *Distributed Algorithm Configuration*, you are presented a more sophisticated version capable of running in a distributed environment and the modifications and technical details necessary to transform a local execution into a distributed one.

Either way, you need to provide the sub-command with a valid scenario. A scenario for pydgga is a directory that contains the configuration for the tuning process, this directory will also be used to store run-time information. For a more detailed explanation please see *pydgga Scenario*.

Sub-command & Arguments

The sub-command to run a *local* version of PyDGGGA is simply called *gga*, and it has the following parameters:

```
pydgga gga -s SCENARIO --slots SLOTS --verbose LEVEL
```

Required

- *-s* | *--scenario*: Path to the scenario directory.

Optional

- *--seed*: Seed for the Pseudo-random number generator. **Default:** 123456
- *--slots*: Number of execution slots, i.e., parallel evaluations. **Default:** # Cores on the machine
- *--executor-verbose*: Verbosity level of the underlying executor library.
- *--verbose*: pydgga verbosity {CRITICAL, ERROR, WARNING, INFO, DEBUG}. **Default:** INFO

Interaction with the target algorithm

There are several ways to let two programs, pydgga and the target algorithm, interact between them. As pydgga only needs to run the target algorithm and gather the result of the evaluation, we deemed that the simplest way is to let them interact via the command line arguments and the standard output streams.

First of all, pydgga will call the target algorithm in a pre-defined way. The first 3 arguments will always be: the *instance* to evaluate the target algorithm, the *cutoff* time in seconds, and the *seed* that the target algorithm must use to guarantee that randomly taken decisions are as close as possible between two runs using the same instance. After this fixed parameters it will pass the parameters specified in the [Algorithm Configuration File](#).

Then, to report the results back to pydgga, the target algorithm will print them as the last line of the standard output using any of the following formats:

- GGA SUCCESS <NUMBER>: To indicate that the instance has been solved and the quality metric (NUMBER).
- GGA SUCCESS RUNTIME: To indicate that the instance has been solved and use the user-time reported by the O.S as the quality metric.
- GGA CRASHED <NUMBER>: To indicate that the evaluation has crashed, but it was expected, and the quality associated to that crash. Ideally a value significantly higher than a very bad, or the worst, possible quality of a *SUCCESS* evaluation (i.e: PAR10 in run-time tuning).
- GGA TIMEOUT <NUMBER>: To indicate that the evaluation has timed out and the quality associated to this event.
- GGA MEMOUT <NUMBER>: To indicate that the evaluation has ran out of memory and the quality associated to this event.
- GGA ABORT: Something unexpected happened that should be fixed. This abruptly stops the search procedure.

Example Glucose

In this example we are going to use pydgga to optimize the SAT solver *glucose4*. You should be able to find this example in *examples/sat/glucose4*, and it should run on any Unix-like operating system, i.e., Linux, MacOS, etc., that have the **GNU coreutils**¹ tools installed.

The provided example already follows the structure of an scenario directory, and it also contains the sources for glucose4. For more information about the scenario files please see [pydgga Scenario](#). In what follows, we only explain the files whose content is more related to the example:

- **build.sh:**

This script builds the SAT solver in the proper location. It does not expect any parameter just *cd* into the scenario directory and run *./build.sh*.

- **wrapper.sh:**

Since we do not want to alter the solver, this script will act as our target algorithm. To get the idea of what the script does, we comment the more significant portions of its code. First, we declare the known exit status values of *glucose4*, this will help us identify if the solver finished properly.

```
3 RET_SAT=10
4 RET_UNSAT=20
5 RET_TIMEOUT=124    # timeout tool man page
```

We also want to test if the glucose4 binary exists and is executable, if it is not this is one of the situations in which we would like to abort PyDGGA execution.

¹ GNU Coreutils Home Page <https://www.gnu.org/software/coreutils>

```

12 if [ ! -f "${BINARY}" ]; then
13     echo "${BINARY} file does not exist"
14     echo "GGA ABORT"
15     exit 1
16 fi
17
18 if [ ! -x "${BINARY}" ]; then
19     echo "${BINARY} is not an executable file"
20     echo "GGA ABORT"
21     exit 1
22 fi

```

If the previous tests pass we continue by extracting PyGGA's fixed parameters <instance> <cutoff> and <seed> from the arguments list:

```

25 instance=${1}
26 cutoff=${2}
27 seed=${3}
28 shift 3

```

Just before running glucose4, we set up the command that will enforce the cutoff constraint (from GNU coreutils).

```

40 uname_out="$(uname -s)"
41 case "${uname_out}" in
42     Linux*)      cutoff_cmd="timeout --foreground ${cutoff}"
43     ;;
44     Darwin*)     cutoff_cmd="gtimeout --foreground ${cutoff}"
45     ;;
46     *)           cutoff_cmd=""
47 esac

```

Finally, we run the solver, recover the exit status and report the appropriate quality value in the format expected by PyDGGA. In this example we use a PAR10 score, thereby our *penalty* quality is set to 10 times the *cutoff*.

```

53 # run solver
54 ${cutoff_cmd} ${BINARY} $@ -rnd-seed=${seed} "${instance}"
55 ret=$?
56
57 echo "" # Make sure the GGA message goes into its own line.
58 case "${ret}" in
59     ${RET_SAT}|${RET_UNSAT}) echo "GGA SUCCESS RUNTIME" ;;
60     ${RET_TIMEOUT}) echo "GGA TIMEOUT ${penalty} it could not be solved :(" ;;
61     *) echo "GGA CRASHED ${penalty}" ;;

```

With all the pieces in place, you can test the command by doing *cd* into the folder that contains the *examples/* directory, and invoking the sub-command as follows

```
pydgga gga -s examples/sat/glucose4
```

If there is no problem, you should get an output similar to:

```

[20-07-2018 13:03:49 - INFO] Command line tool version 0.2.1
[20-07-2018 13:03:49 - INFO] PyDGGA version 1.5.8 beta
[20-07-2018 13:03:49 - INFO]
[20-07-2018 13:03:49 - INFO] Product licensed to log [Logic Optimization Group]
[20-07-2018 13:03:49 - INFO] Subscription valid until: ...
[20-07-2018 13:03:49 - INFO]
[20-07-2018 13:03:49 - INFO] PyDGGA invoked with commandline
[20-07-2018 13:03:49 - INFO] gga -s examples/sat/glucose4 --slots 1

```

(continues on next page)

(continued from previous page)

```
[20-07-2018 13:03:49 - INFO]
[20-07-2018 13:03:49 - INFO] ===== Scenario =====
[20-07-2018 13:03:49 - INFO] wrapper_exe = sh
[20-07-2018 13:03:49 - INFO] wrapper_name = wrapper.sh
[20-07-2018 13:03:49 - INFO] population = 25
[20-07-2018 13:03:49 - INFO] generations = 100
[20-07-2018 13:03:49 - INFO] min_generations = 20
...
[20-07-2018 13:03:49 - INFO] ===== Instance Selection Strategy =====
[20-07-2018 13:03:49 - INFO] # Instances: 41
[20-07-2018 13:03:49 - INFO] # Instances Groups: 4
[20-07-2018 13:03:49 - INFO] # Instances on first generation: 5
[20-07-2018 13:03:49 - INFO] Use all instance at generation: 20
[20-07-2018 13:03:49 - INFO] Instances root directory: examples/sat/instances
[20-07-2018 13:03:49 - INFO]
[20-07-2018 13:03:49 - INFO] ===== Results Cache =====
[20-07-2018 13:03:49 - INFO] # Pre-loaded results: 0
...
[20-07-2018 13:03:49 - INFO] Running GGA
...
```

To get notified every time an evaluation finishes, run the command with: `--verbose DEBUG`.

2.1.2 Distributed Algorithm Configuration

As mentioned previously, there are two ways of running PyDGGA using the *PyDGGA Command Line Tool*. This section corresponds to the distributed execution, capable of exploiting the computing resources from different machines. Previously, in *Local Algorithm Configuration*, you were presented the command that runs locally in one machine, we recommend starting there before continuing with the distributed version.

The content of this section assumes that the reader is already familiar with running pydgga locally. Thereby, this section will focus only on the additional steps to run pydgga in a distributed environment. In fact, pydgga has been designed so that you can reuse an scenario that worked previously on a local machine and run it in a distribute manner by just adding some arguments to the command line.

Note: The distributed implementation only provides the means for distributing the workload, other resources such as the binary file or the instances, must be accessible on the machines running the workers via other mechanisms, i.e., shared file system, etc.

Sub-command & Arguments

The sub-command to run a *distributed* version of PyDGGA is called `dgga`, and it has the following arguments

```
pydgga dgga -s SCENARIO --seed INT --slots SLOTS --verbose LEVEL
              --port PORT --worker-script WSCRIPT --num-workers NUMW
```

Required

- `-s` | `--scenario`: Path to the scenario directory.

Optional

- `--seed`: Seed for the Pseudo-random number generator. **Default:** 123456
- `--slots`: Number of execution slots, i.e., parallel evaluations. In the distributed version, this option is passed to the `--worker-script`, which can use it to determine the number of slots to allocate for each worker. **Default:** #CPU cores on the master node

- `--port`: Port where the master will listen for worker connections, if not specified lets the O.S pick the port. If you let the O.S pick the port you will need a `--worker-script` at least to echo the port value. **Default:** 0
- `--worker-script`: Path to a script responsible of starting new workers. Upon invocation the script will receive a list of arguments: first the name of the scenario directory, then the number of slots to allocate, and the rest must be forwarded to pydgga. The exit status must be 0 on success. **Default:** *<empty string>*
- `--num-workers`: Number of desired workers. This is the number of workers that the tool will try to keep automatically, using the `--worker-script`, but more workers can be started manually. **Default:** 1
- `--executor-verbose`: Verbosity level of the underlying executor library.
- `--verbose`: pydgga verbosity {CRITICAL, ERROR, WARNING, INFO, DEBUG}. **Default:** INFO

As in *Local Algorithm Configuration*, the only mandatory arguments is the scenario, the rest of the arguments purpose is automatize the process of running worker processes.

Options `--slots` and `--num-workers` are only effective if option `--worker-script` is set to point to a valid script or executable file. Also, `--port` lets the user pick the port the master will listen to in **all** the available interfaces (bind to `*:<port>`), if there is another program bound to any interface on the same TCP port, the master will not start.

Note: The distributed version is build on top of the local version, thereby the execution of the target algorithm is conducted as if a local gga was running on the remote machine. Meaning all paths must remain valid in the remote machine (the shared file system must be mounted on the same path).

Example Glucose (local)

This example is build upon the example in *Example Glucose*, if you have not done it yet please read that example before continuing with this one. In what follows we assume that the reader has managed to run local example and we will now run it using the distributed version of PyDGGA.

In fact, we are only going to add one file to the ones already present, a script that will manage the submission of worker processes. This script is not even necessary, we could just simply run the workers manually, but it is more practical to delegate this task to be handled automatically.

For demonstration purposes, in this example we will start the workers on the same machine the master is running. The process to run them in a computing cluster environment, with a shared file system, only requires tweaking the script. Also, you can find useful templates for the worker submission script in section *PyDGGA Templates*.

When pydgga calls the script responsible of starting the workers, it passes the following parameters:

1. The session name.
2. Number of slots (the value of the *slots* argument for the master).
3. Other parameters required to start the worker, such as the port and IPs the worker should use to connect to the master.

The first two are useful when allocating jobs in a batch-queueing system, you can use them to: give the job a name, specify the number of CPUs required for the job and specify the file names for *stdout* and *stderr*. The following listing shows a *bash* script that runs workers on the same machine using *nohup* and thereby does not use the number of slots.

```

1  #!/usr/bin/env sh
2
3  # Current working directory
4  CWD=`pwd`
5
6  # How pydgga must be invoked in your system may vary
7  # here we assume it is accessible in the $PATH
```

(continues on next page)

(continued from previous page)

```

8 PYDGGA="pydgga dggaw"
9
10 name=${1} # scenario name
11 slots=${2} # number of slots (unused)
12 shift 2 # remove the first two parameters from ${@}
13
14 # Create the output file name using the scenario's name
15 t_file=$(mktemp "${CWD}/${name}-XXXXXXXXXX")
16 echo "Output file: ${t_file}"
17
18 # Finally, start the worker using nohup and redirect its output.
19 nohup ${PYDGGA} ${@} > "${t_file}" &
20
21 exit 0

```

Now, before executing the distributed version of PyDGGA, copy the script shown above in a file named *examples/sat/glucose4/start-worker.sh* and give it execution permissions (*chmod +x*). Then, you can invoke the distributed PyDGGA by doing *cd* into the folder that contains the *examples* directory and using the command (you can change the number of slots, workers or the port if you want):

```

pydgga dggaw -s examples/sat/glucose4 \
              --slots 2 --num-workers 2 \
              --worker-script examples/sat/glucose4/start-worker.sh

```

If everything works fine, the worker script will be called twice by PyDGGA and each worker will be able to run two evaluations simultaneously. Finally, you should get an output similar to the one you get when running the local version of GGA

```

[20-07-2018 16:02:10 - INFO] Command line tool version 0.2.1
[20-07-2018 16:02:10 - INFO] PyDGGA version 1.5.8 beta
[20-07-2018 16:02:10 - INFO]
[20-07-2018 16:02:10 - INFO] Product licensed to log [Logit Optimization Group]
[20-07-2018 16:02:10 - INFO] Subscription valid until: ...
[20-07-2018 16:02:10 - INFO]
[20-07-2018 16:02:10 - INFO] PyDGGA invoked with commandline
[20-07-2018 16:02:10 - INFO] dggaw -s examples/sat/glucose4/ --slots 1 --num-
↪workers 2 --worker-script examples/sat/glucose4/start-worker.sh
[20-07-2018 16:02:10 - INFO]
[20-07-2018 16:02:10 - INFO] ===== Scenario =====
[20-07-2018 16:02:10 - INFO] wrapper_exe = sh
[20-07-2018 16:02:10 - INFO] wrapper_name = wrapper.sh
[20-07-2018 16:02:10 - INFO] population = 25
...
[20-07-2018 16:02:10 - INFO] Installing signal handlers
[20-07-2018 16:02:10 - INFO] Initializing GGA
[20-07-2018 16:02:10 - INFO] Starting executor
Output file: /home/$USER/Desktop/-5mcAWjbnu2
[20-07-2018 16:02:10 - INFO] Worker submission [OK]
Output file: /home/$USER/Desktop/-BILXuUN8c
[20-07-2018 16:02:10 - INFO] Worker submission [OK]
[20-07-2018 16:02:10 - INFO] Running GGA
...

```

Like in the local execution, to get notified every time an evaluation finishes add *--verbose DEBUG* to the command line.

Example Glucose (SGE)

In the previous example we have already introduced all there is to run PyDGGA in a distributed environment, what we did not talk about is how to do so in a real scenario. Here we modify the previous example to run on a cluster that uses Sun Grid Engine, Open Grid Engine or any other batch-queue system based on them.

On this systems, jobs are submitted for execution using a tool called *qsub*, which receives the script to run, usually a Bash, sh or Python file with the proper *shebang*. Another alternative that we will use in this example is passing the script that *qsub* has to run via its standard input.

The first step is defining how we want to run the job on SGE. To do so, we first need to know some information that is specific to our environment, such as the *queue* and the *parallel environment* we should use. This information will vary from system to system and we cannot provide a full description of all the additional information that may be required on your system configuration. For this particular example, this is the configuration we would like to achieve:

- Run on a queue named: **hpc1.q** (it may be different on your cluster)
- Run on the symmetrical multiprocessing parallel environment: **smp** (this is usually a default environment available on any SGE installation)
- Save the master output (stdout & stderr) into the user's HOME directory.
- Save the worker output (stdout & stderr) into a directory *workers* inside the user's HOME directory.

With the previous configuration in mind we first write the script that will submit workers and mark the file as executable (*chmod +x*).

```
#!/usr/bin/env sh

# How PyDGGA must start in your system may vary (assumes it is in the PATH)
PYDGGA="pydgga dggaw"

name=${1} # scenario name
slots=${2} # number of slots
shift 2 # remove the first two parameters from ${@}

# Path for the stdout and stderr files
olog="${HOME}/workers"
elog="${HOME}/workers"

mkdir -p "$olog" # Ensure the logs directory exists

# Command to run PyDGGA
cmd="${PYDGGA} ${@} --verbose WARNING"

# Passing the script/command to run via the standard input
echo "$cmd" | qsub -pe smp ${slots} -cwd -q hpc1.q \
    -N "${name}" -o "$olog" -e "$elog"

exit 0
```

Now we are ready to write the script that will submit PyDGGA's master process. This script will be a bit different due to the way *qsub* receives its arguments but both accomplish the same objective.

```
#!/usr/bin/env sh

## Comments that start with ## are interpreted by *qsub* as parameters
## given on the command line.

#$ -S /bin/bash
#$ -cwd
#$ -q hpc1.q
#$ -pe smp 1
```

(continues on next page)

(continued from previous page)

```

#$ -N Glucose4-qsub-example
#$ -o $HOME/$JOB_NAME.o$JOB_ID
#$ -e $HOME/$JOB_NAME.e$JOB_ID

## Limit the job to run for 2 days
#$ -l s_rt=48:00:00

# modify the scenario path according to your current working directory.
SCENARIO="examples/sat/glucose4"

# modify the worker script path according to where you have save it.
WORKER_SCRIPT="./start-worker-qsub.sh"

NUM_WORKERS=4
SLOTS_PER_WORKER=4

## Execute dgga
pydgga dgga -s "${SCENARIO}" \
             --slots ${SLOTS_PER_WORKER} \
             --worker-script "${WORKER_SCRIPT}" \
             --num-workers ${NUM_WORKERS}

```

Finally, all there is to do is submit the master process using *qsub* and let *pydgga* use the "start worker" script to do the rest.

```
qsub pydgga-glucose4-qsub.sh
```

Note: On some systems, for stability and security reasons, not all nodes can be used to submit jobs. On these systems the master process should be run as a background process in the frontend or login node of the cluster (usually using the *nice* command to avoid interfering with other users that may use the same login node).

2.1.3 Distributed Algorithm Configuration (Worker)

This sub-command of *pydgga* starts it in *dgga* worker mode to handle evaluations from another *pydgga* running in master mode (*dgga*). The only purpose of the worker program is conducting evaluations when the master requests it. It outputs its own separated log where it prints the received tasks, the computed results and other events for debugging purposes. It operates using TCP connections and will stop automatically if the connection fails to start, is abruptly terminated or the master requests its shutdown.

Note: As mentioned in section *Distributed Algorithm Configuration*, the worker only provides computing capabilities to the master process, the target algorithm and the instances must be accessible using other mechanisms, i.e., shared file system, etc.

Note: If the reader intends to run the worker manually it is important to be able to determine the IP and PORT the master uses to listen for incoming connections.

Sub-command & Arguments

The sub-command to run pydgga in worker mode is called `dggaw`, and it has the following arguments

```
pydgga dggaw --slots SLOTS --address ADDRESS --port PORT --verbose LEVEL
```

Required

- `--address`: Address to connect to. This argument can be used more than once to specify all the addresses the worker must try to connect to.
- `--port`: Port where the master is listening for worker connections.
- `--slots`: Number of execution slots, i.e, parallel evaluations, that the worker supports.

Optional

- `--executor-verbose`: Verbosity level of the underlying executor library. **Default:** INFO
- `--verbose`: pydgga verbosity {CRITICAL, ERROR, WARNING, INFO, DEBUG}. **Default:** INFO

Usually this command is run from a *start worker* script and all the necessary parameters are provided by pydgga.

2.2 PyDGGA File Formats

In this section we present you the file formats that pydgga uses for both input (configuration, restore state, etc) and output (save state, cache, etc). These files not only help pydgga internally, they also provide the user with an easy way of configuring pydgga without having to retype the same configuration over and over again. Moreover some files can be used outside pydgga to compute statistics, recover evaluation results, etc.

2.2.1 pydgga Scenario

A pydgga scenario is the main building block of its configuration. Basically it is a directory with some special files that contain different parts of the configuration. Then, when running, pydgga creates a special *run_X* directory that it uses to store the cache and other run-time specific data.

So, what makes a directory an scenario? Basically, having the following files in a directory makes it a scenario:

- **conf.xml** This file describes the parameter structure of the target algorithm as a tree and also contains user provided configurations, for a more detailed explanation see [Algorithm Configuration File](#).
- **instances.txt** File with the lists of instances that pydgga uses to evaluate the different configurations. See [Instances File](#).
- **settings.txt** File that contains the configuration of the different components pydgga uses to conduct the tuning process. The following sub-sections are dedicated to this file.
- **wrapper file** File that pydgga executes to evaluate the target algorithm, it may be the target algorithm itself.

Settings File

The settings file is just a plain text file that follows a small set of very simple formatting rules:

- If the first non-blank character is `#`, the line is considered a comment and will be ignored during parsing.
- Blank lines or lines that only contain whitespaces (or any other type of blank character) will also be ignored.
- Any other line must be divided in 3 parts, the name of the property, the separator (`=`), and the value of the property. Whitespaces at the ends of the property name and the property value will be removed.

The following listing illustrates the previous rules:

```

1 # This line is a comment and the next is blank. Both will be ignored.
2
3 property1 = value for property 1
4 property2 = 3.0
5 # ... and so on

```

Properties

The following list contains the names of the properties accepted by pydgga, its type (*{int}*, *{float}*, *{str}*), range or possible values (between *[]*, if applies), and a brief explanation of what they affect.

Note: Properties of type *{bool}* accept the following truth values:

- As **True**: 'true', 't', 'yes', 'y'
 - As **False**: 'false', 'f', 'no', 'n'
-

- **wrapper_exe** *{str}*: The program that pydgga must use to execute the wrapper program, specified by the option **wrapper_name**. If the wrapper program is an executable this property is not necessary. Its main purpose is being able to run scripts in OS that do not support executable text files such as Windows.
- **wrapper_name** *{str}*: Name of the program/file that pydgga must execute to evaluate the target algorithm, this program/file must be inside the scenario directory.
- **population** *{int}* [*>0*]: Desired population size. GGA will do its best to keep the population stable around the specified value. Nonetheless it will oscillate about 15-20% up and down.
- **generations** *{int}* [*>0*]: Maximum number of generations to run. Once GGA reaches this number of generations it will stop.
- **min_generations** *{int}* [*>=0*]: Minimum number of generations to execute. This value is useful when the tuning process converges too fast and you wish to let pydgga keep exploring. It does not affect the maximum number of generations nor any other limit (# evaluations/real time limit, etc).
- **max_age** *{int}* [*>0*]: Lifespan of an individual/genome. An individual will survive into the next generation only if its age does not exceed this value. The winner of the generation is immune to death by age and will always be part of the next generation.
- **rand_replace_prob** *{float}* [*>=0*]: Probability of replacing a non-competitive individual by a new one with a randomly generated genome. The default value of 0 is usually good as recombination and mutation are enough to escape local optima, nonetheless some target algorithms may benefit from changing this value.
- **eval_group_size** *{int}* [*>0*]: Desired number of individuals to compete against each other. This value is the mini-tournament size in previous versions of GGA. Since this property defines the number of evaluation groups, setting a high value will reduce the diversity (less winners), as a rule of thumb we usually set this value to have at least 5 winners per generation.
- **eval_time_limit** *{int}* [*>0*]: Time limit for each genome/instance evaluation. This limit is not enforced by pydgga but rather passed down the user provided execution wrapper as the *cutoff* parameter. It is important that the wrapper tries to honor this limit since pydgga will use it to schedule the evaluations.
- **tuner_rt_limit** *{int}* [*>0*]: Real/Wall-clock time limit of the overall tuning process. As with the number of generations, pydgga will stop when the specified amount of time has passed. **WARNING:** the maximum value of this parameter is system dependant, but it should never be less than 2^{31} .
- **tuner_evals_limit** *{int}* [*>0*]: Maximum number of evaluations that GGA can conduct during the tuning process, it will stop once this limit is reached.
- **winners_percentage** *{float}* [*0, 1*]: Percentage of winners to extract from each evaluation group. Usually one winner per evaluation group is preferred, but if there are only a few evaluation groups it may be necessary to get more winners from each group to preserve diversity.

- **mutation_probability** {float} [0, 1]: Probability of mutating the parameters of the offspring. Parameters that only have one value do not participate in the mutation process, thereby if your algorithm has 10 parameters but 2 only have one value and you set a 10% chance of mutation, this 10% only affects the other 8 parameters.
- **sigma_percentage** {float} [0, 1]: Mutations are implemented using a gaussian distribution. This value is used to determine the standard deviation for the distribution: $stdev = \# \text{current_value} * \text{sigma_percentage}$.
- **crossover_operator** {str} [gga]: The operator that must be used when recombining a competitive and a non competitive genome to create a new one.
- **crossover_probability** {float} [0, 1]: When computing the crossover of two genomes, this is the probability of changing to use information from the other genome. A lower value will generate genomes that try to stick to one of the parents while including some parts of the other parent.
- **use_elite_group** {bool}: This property tells PyDGGA to keep alive and evaluate the absolute winner of all generations in a separated group.
- **objective** {str} [avgX, sumX, parX]: The objective function to evaluate the performance of the genomes. As their name suggest the objective functions sum or average the value reported by the target algorithm, the X at the end is optional and must be an integer number used to penalize those evaluations that report a status different than *SUCCESS*. *par* behaves like *avg* but penalizes evaluations that exceed the maximum evaluation time even if they report *SUCCESS*.
- **cost_min** {float}: Best possible quality of an evaluation. This value is used internally for cancellation, scheduling, etc. Ignored if the objective is *par*.
- **cost_max** {float}: Worst possible quality of an evaluation. This value is used internally for cancellation, scheduling, etc. Ignored if the objective is *par*.
- **cost_tolerance** {float}: Minimum difference between two evaluation values to consider them different.
- **cancel** {str} [none, cost, incumbent]: In order to save time when evaluations are deemed unnecessary, PyDGGA can cancel them. From the possible options:
 - none: will never cancel.
 - cost: will cancel only if the cost of the genome being tested for cancellation exceeds the best cost among those that has been evaluated on all the instances of the mini-tournament.
 - incumbent: similar to cost but does not wait until a genome has been evaluated on all the instances, it compares against the best cost among those that have run more instances so far.
- **cancel_min_evals** {int} [>=1]: Minimum number of evaluations required to perform cancellation analysis (only affects aggressive strategies such as **incumbent**).
- **instance_selector** {str} ['rlinear', 'ilinear']: Instance selection strategy. Both strategies increase the number of instances selected at each generation, but *rlinear* picks them at random while *ilinear* picks the same used in the previous generation and adds some more.
- **instances_dir** {str}: Directory that contains the instances specified in the instances file. This directory will be prepended to the instances when passing them to the target algorithm.
- **instances_min** {int} [>0]: Number of instances to use in the first generation.
- **instances_max** {int}: Maximum number of instances that can be used to evaluate the target algorithm. A value of 0 is replaced by the number of instances in the instances file. A negative value is a hint to compute the appropriate value to increase that number of instances at each generation, i.e: a value of -3 will be replaced by a value that adds 3 instances at each generation.
- **instances_gen_max** {int} [>0]: Generation at which reach *instances_max*. After this generation *instances_max* will be used for the remaining generations.
- **seed** {int}: Seed value for the Pseudo-Random number generator.

2.2.2 Algorithm Configuration File

In this section we will explain the different sections of the algorithm configuration file. As for now, the only format supported for this file is XML, and its structure is defined using XSD-schema. First we will introduce the different sections and then move to an example.

XML Sections

The XML file is divided in a sort of "virtual" sections that must be defined in the specified order, failing to do so will not comply with the XSD-schema and thereby fail to load the file. The sections in the proper order are:

1. Parameter Tree
2. Seed Genomes
3. Parameter Constraints

All the specification happens within the `<algconf>``</algconf>` tags, and to explain each of these sections, we will focus on the following example:

```

1 <algconf>
2   <node type="and" name="root" domain="[0, 0]" ignore="true">
3     <node type="and" name="param1" domain="[0, 4]" prefix="-param1="/>
4     <node type="or" name="param2" domain="{a,b,c}" prefix="-param2=">
5       <node type="and" name="param3" domain="[5.5, 7.5]"
6         or-domain="{a}" prefix="-param3="/>
7       <node type="and" name="param4" domain="[10, 20]"
8         or-domain="{a,b}" prefix="-param4="/>
9       <node type="and" name="param5" domain="{cat1,cat2}"
10        or-domain="{a,c}" prefix="-param5="/>
11     </node>
12   </node>
13
14   <seedgenome>
15     <param name="root" value="0"/>
16     <param name="param1" value="3"/>
17     <param name="param2" value="c"/>
18     <param name="param3" value="5.654"/>
19     <param name="param4" value="12"/>
20     <param name="param5" value="cat2"/>
21   </seedgenome>
22
23   <constraints>
24     <cstr><![CDATA[not param1 < 3 or param2 in ["a", "b"]]]</cstr>
25     <cstr><![CDATA[5.5 <= param3 <= 6 and param4 < 15]]</cstr>
26   </constraints>
27 </algconf>

```

Parameter Tree

The next part of the file (`<node>``</node>`) specifies the parameter tree itself. The tree is specified by a single node corresponding to the root of the tree and each `node` tag may have any number of nodes underneath it. Each `node` tag may have one of two types ("or" or "and"), which will be described later, the name of the parameter and the domain of the parameter, which can be categorical, discrete or continuous. Additionally, some nodes may also define the or-domain, that defines for which values of the parent "or" node that parameter must be included in the configuration.

The syntax of the domain of the nodes defines whether they are categorical, discrete or continuous. Categorical domains are enclosed in brackets `{}` and each value is separated by a comma `,`, also white spaces between the comma and the value are preserved. Discrete domains are enclosed in square brackets `[]` that contain exactly two **integer** values separated by a comma `,`, the first value is the lower bound of the domain and the other value is the

upper bound, both included in the domain. Continuous domains are defined as Discrete domains but at least one of the values, lower/upper bound, must be a floating point number, such as 1.0 or 2.3.

An "and" node indicates that the child nodes should all be present whenever the parent node is present in the parameter settings of a configuration. In contrast, an "or" node means that only some nodes should be included in the setting. In other words, or nodes allow users to select which parameters must be included when another parameter takes a specific value. In the example above, param2 has three settings: a, b and c. Setting a is associated with all the branches, setting b with param4 branch and setting c with param5 branch. Thus if params 1 to 5 take the following values: [3, cm 5.654, 12, cat12] as in the seed genome, the following command would be executed

```
./example_solver (instance) (seed) -param1=0 --param2=c -param5=cat2
```

Note that param3 and param4 are not given on the command line because their branches were not selected.

Seed Genomes

This part is optional and is defined using as many `<seedgenome>` as needed, in the example above there is only one seed genome but you can have as many as you need. Each seed genome specifies settings of parameters to insert into the initial population, each `<param ... />` tag specifies the name of a parameter and the value for it to take in the configuration. When defining a seed genome, all parameters must be specified, even those that are not used, such as the tree root. Also, needless to say that the values in the seed genome must be compatible with the types of the parameters, i.e., not using string values on numerical parameters.

Constraints

The final part of the file (`<constraints>`) specifies which parameter combinations are forbidden to be used together. Each `<ctr>` block specifies a Python like boolean expression involving a set of parameters that must be True in order to consider a configuration valid.

In the above example if param1 is less than 3 then param2 must be either "a" or "b", but not "c", and if param3 is between 5.5 and 6 param4 must be less than 15.

2.2.3 Instances File

This file is pretty straight forward, it is a plain text file where each line contains a seed and an instance. Ideally the seeds are there so that the results of the target algorithm do not depend on the state of the pseudo-random number generator. Nonetheless, how the seeds are used, if they are used at all, depend on the target algorithm or its wrapper. The following listing is a portion of the file included in the glucose example.

```
24727 bejing/2bitadd_10.cnf.gz
11545 bejing/2bitadd_11.cnf.gz
11046 bejing/2bitadd_12.cnf.gz
26951 bejing/2bitcomp_5.cnf.gz
24061 bejing/2bitmax_6.cnf.gz
11030 bejing/3bitadd_31.cnf.gz
```

Additionally, clusters of instances can be specified by separating them in the instances file using a blank line.

```
12409 bmc/bmc-ibm-4.cnf.gz
724 bmc/bmc-ibm-5.cnf.gz
27426 bmc/bmc-ibm-6.cnf.gz
28101 bmc/bmc-ibm-7.cnf.gz

15819 phole/hole6.cnf.gz
30670 phole/hole7.cnf.gz
```


2.3 PyDGGA Templates

In this section we provide some useful templates to ease the process of using PyDGGA from scratch, from target algorithm wrappers to worker start scripts.

2.3.1 Worker Script (local)

Bash

This bash script receives the parameters from PyDGGA and starts a detached worker using *nohup*. You may need to change some parts or extend the script+ to fit your system and other requirements you may have.

```
#!/usr/bin/env sh

# Current working directory
CWD=`pwd`

# How pydgga must start in your system may vary
# here we assume it is accessible via $PATH
PYDGGA="pydgga dggaw"

# Extract fixed parameters
name=${1}    # session name
slots=${2}   # number of slots
shift 2      # remove the parameters from ${@}

# Print some info (once everything work you can comment this code)
echo "Requested number of slots ${slots}, this may be useful to allocate"
echo "resources in computer clusters, etc."
echo ""
echo "PyDGGA parameters:"
echo "${@}"

# In order to save the output and avoid name clashes we relay on
# mktemp to create a new file name.
t_file=$(mktemp "${CWD}/${name}-XXXXXXXXXX")
echo "Output file: ${t_file}"

# Finally, start the worker using nohup and redirect its output.
nohup ${PYDGGA} ${@} > "${t_file}" &

exit 0
```

2.3.2 Worker Script (qsub)

Bash

This bash script receives the parameters from PyDGGA and submits a worker using *qsub*. Some parts of the template must be adapted to fit your system and your requirements, such as the queue to submit to or the memory and time limits. Moreover, you may also need additional constraints or modify the environment.

```
1  #!/usr/bin/env sh
2
3  # How pydgga must start in your system may vary
4  # here we assume it is accessible via $PATH
5  PYDGGA="pydgga dggaw"
6
7  # Some of your system specific configuration goes here
```

(continues on next page)

(continued from previous page)

```

8 QUEUE="yourqueue.q"
9 PENV="smp"
10 MEM_LIMIT="35840M" # 35 GB
11 RT_LIMIT=172800    # 2 Days
12
13 # Extract fixed parameters
14 name=${1}          # session name
15 slots=${2}         # number of slots
16 shift 2            # remove the parameters from ${@}
17
18 # Print some info (once everything work you can comment this code)
19 echo "Requested number of slots ${slots}, this may be useful to allocate"
20 echo "resources in computer clusters, etc."
21 echo ""
22 echo "PyDGGA parameters:"
23 echo "${@}"
24
25 olog="/path/to/stdout/file_or_directory"
26 elog="/path/to/stderr/file_or_directory"
27
28 mkdir -p "$olog"    # Make sure the stdout directory exists
29 mkdir -p "$elog"    # Make sure the stderr directory exists
30
31 cmd="${PYDGGA} ${@} -v INFO"
32
33 echo "${cmd}" | qsub -V -cwd -pe ${PENV} ${slots} \
34                   -l h_core=1G -l h_vmem=${MEM_LIMIT} \
35                   -l h_rt=${RT_LIMIT} -q ${QUEUE} \
36                   -N ${name} -o "${olog}" -e "${elog}"
37
38 exit 0

```

2.3.3 Worker Script (ccsalloc)

ccsalloc is a command of the OpenCSS¹ batch-queueing system.

Bash

This bash script receives the parameters from PyDGGA and submits a worker using *ccsalloc*. Some parts of the template must be adapted to fit your system and your requirements, such as memory and time limits. Moreover, you may also need additional constraints or modify the environment.

```

1 #!/usr/bin/env sh
2
3 # How pydgga must start in your system may vary
4 # here we assume it is accessible via $PATH
5 PYDGGA="pydgga dggaw"
6
7 # Expected input: <name> <slots> <dgga_parameters>
8 name=${1}
9 slots=${2}
10 shift 2
11
12 # Job configuration
13 MEMORY=`echo "3072 * ${slots}" | bc` # 3072MB for each CPU
14 RT_TIME=48h

```

(continues on next page)

¹ <https://wikis.uni-paderborn.de/pc2doc/OpenCCS>

(continued from previous page)

```
15 OUT_LOG="path/to/stdout/file/${name}.o%reqid"
16 ERR_LOG="path/to/stderr/file/${name}.e%reqid"
17
18 mkdir -p `dirname "${OUT_LOG}"`
19 mkdir -p `dirname "${ERR_LOG}"`
20
21 # Allocate job
22 ccsalloc
23     --res rset=1:ncpus=${slots}:mem=${memory}m \
24     --walltime ${RT_TIME} \
25     --name ${name} \
26     --stdout ${OUT_LOG} --stderr ${ERR_LOG} \
27     -- ${PYDGGA} ${@}
28
29 RET=$?
30 exit "${RET}"
```