TinyverseGP User Guide

1 Introduction

TinyverseGP is a modular cross-domain benchmarking framework for genetic programming (GP). The current version provides modules for tree-based and graph-based GP as well as modules to apply it to four different problem domains. TinyverseGP follows the vision to establish a flexible and light-weight benchmarking framework that is easy to extend and can be used to achieve more comprehensive benchmarking results in GP.

On the representation level, TinyverseGP supports:

- **Tree-based Genetic Programming** (TGP) (also known as Koza-style): the programs are represented as trees. This version supports multi-tree
- **Cartesian Genetic Programming** (CGP): the programs are represented graphs and naturally encodes multiple outputs with shared components.

On the application level, Tinyverse supports the following problem domains:

- **Symbolic Regressio**n
- **Logic Synthesis**
- **Policy Learning**
- **Program Synthesis**

The codebase is written in Python trying to keep the requirements to a minimal. The codebase is organized into different modules, each of which implements a different representation for Genetic Programming. The following representations are currently implemented:

2 Requirements

The current version supports Python 3.9 and higher. To install the requirements it is suggest to run:

```
python3 -m venv env
. env/bin/activate
pip3 install -r requirements.txt
```

The `requirements.txt` lists the following dependencies needed to run TinyverseGP:

```
Box2D==2.3.10
pygame==2.6.1
gymnasium==1.0.0
numpy>=1.21.0
dd>=0.6.0
swig==4.3.0
sympy==1.13.3
requests==2.32.3
ale_py==0.10.1
```

Please note that to run Gymnasium environments that use `Box2D`, it has to be installed manually via `pip install Box2D`. The Gymnasium Atari Learning Environment has to be installed via PyPI `pip install ale-py`.

3 Repository structure

The TinyverseGP repository is organized as follows:

- src/gp: contains the core implementation of the different representations.
  - `tiny_tgp.py`: implementation of Tree-based Genetic Programming (TGP).
  - `tiny_cgp.py`: implementation of Cartesian Genetic Programming (CGP).
  - `tineverse.py`: the abstract classes for GP, Config, Hyperparameters, and Function set.
  - `functions.py`: the standard set of functions currently supported. `problem.py`: the abstract class for the problem to be solved. It includes the example based problem (black-box), policy search, and program synthesis. It includes the example based problem (black-box), policy search, and program synthesis.
  - `loss.py`: currently supported loss functions.
- `src/benchmark`: contains the benchmark problems and datasets.
  - `symbolic_regression/sr_benchmark.py`: sample symbolic regression benchmark
  - `symbolic_regression/srbench.py`: interface to the SRBench benchmark suite.
  - `logic_synthesis/ls_benchmark.py`: sample logic synthesis benchmark problems.
  - `policy_search/policy_evaluation.py`: interface to the gymnasium environment.
- `src/examples`: examples on how to use the different benchmarks. problems.

4 Benchmarks

4.1 Symbolic Regression

Two ways are provided run symbolic regression benchmarks with TinvyverseGP. The first way is to use the `SRBenchmark` class in `benchmark/symbolic_regression/sr_benchmark`. The class already provides functionality to generate a dataset that is sampled from uniform

distribution via the `dataset_uniform` method as well as some simple benchmark functions. commonly used benchmark functions soon. The second way is to use the interface to SRBench[1] in the `srbench` module. The `src/gp/loss` module provides loss functions that are commonly used for evaluating symbolic regression models. An example given in `examples/symbolic_regression/test_bench`.

4.2 Logic Synthesis

The repository of TinyverseGP includes the benchmark problems of the General Boolean Function Benchmark Suite (GBFS) in the `data` folder. The respective benchmarks are stored in `PLU/PLA`[2], `TT` and `BLIF`[3] format. Additionally, we included the Boolean Benchmark Tools (BBT)[4] that have been provided for the use of GBFS. BBT offers interfaces for reading `PLU/PLA` and `TT` files that are then represented with truth tables. Bitwise hamming distance is commonly used to evaluate candidate expressions or circuits. Am implementation is provided in the `src/gp/loss` module. An example on how to use a benchmark that is represented with a `PLU/` file is provided in the `examples/logic_synthesis` folder.

4.3 Policy Learning

We provide a policy learning problem class in `src/gp/problems` that is connected with an agent class `GPAgent` to evaluate candidate policies in the respective gym environment. The GPAgent class therefore serves as a bridge between the environment and the GP system. To use a certain environment of Gymnasium, the observation space and action space has to be adjusted to the GP configuration. For instance, the number of *variable* nonterminal symbols can be obtained from the size of the observation space. The prediction, either discrete or continuous, of a action that is supposed to be performed by the agent can be obtained from the action space. The size of the observation and action space can be used to configure the number of inputs and outputs for Cartesian GP. The module `src/benchmark/policy_search/pl_benchmark` provides a benchmark class `PLBenchmark` which can be used to obtain the size of the observation and action space for the respective environment. This class also handles reprocessing for the Gymnasium Arcade Learning Environments (ALE)[5]. An example of how to use ALE environments with TinyverseGP is provided in `examples.policy_learning.test_cgp_pl_ale`.

4.4 Program Synthesis

We provide a benchmark class to handle some coding problems from Leetcode.com[6], a popular coding interview preparation website. The corresponding program synthesis problem that is implemented in `src/gp/problem` and used to solve this kind of problem can be boiled down to a binary classification problem.

[1] https://github.com/cavalab/srbench/tree/master
[2] https://ddd.fit.cvut.cz/www/prj/Benchmarks/pla_c.html
[3] https://course.ece.cmu.edu/ ee760/760docs/blif.pdf
[4] https://github.com/RomanKalkreuth/boolean-benchmark-tools
[5] https://ale.farama.org/environments/
[6] https://leetcode.com/

5 Configuration and Hyperparameters

The following listing give an overview about the general configuration for TinyverseGP as well as the hyperparamter settings for tree-based and cartesian GP.

```
num_jobs:int                # Number of runs in the experiment
max_generations: int        # Maximum generation count per run
minimizing_fitness: bool    # Minimizing or maximizing problem
ideal_fitness: float        # Ideal fitness value to reach
silent_algorithm: bool      # No reporting during a run
silent_evolver: bool        # No reporting after a run
minimalistic_output:bool    # Only best fitness and number of fitness is printed
report_interval: int        # Interval setting for immediate results reporting
max_time: int               # Timelimit for each run
```

Listing 1: General configuration for GP experiments

```
pop_size: int               # Size of the population
max_size: int               # Maximum size of the program trees
max_depth: int              # Maximum depth of the p
mutation_rate: float        # Mutation strength
cx_rate: float              # Crossover probabilty
tournament_size: int        # Selection pressure control
```

Listing 2: Hyperparameters for tree-based GP

```
num_functions: int          # Number of functions in the function set
max_arity: int              # Maximum arity in the function set
num_inputs: int             # Number of inputs nodes
num_outputs: int            # Number of output nodes
num_function_nodes: int     # Number of function nodes
```

Listing 3: Hyperparameters for Cartesian GP

6 Testing

Test scripts are provided to apply an instance of TinyverseGP to each of the supported problem domain. In order to test TinverseGP, you can use one of the following commands:

```
python3 -m examples.symbolic_regression.test_cgp_sr
python3 -m examples.symbolic_regression.test_tgp_sr
python3 -m examples.logic_synthesis.test_cgp_ls
python3 -m examples.logic_synthesis.test_tgp_ls
python3 -m examples.policy_learning.test_cgp_pl
python3 -m examples.policy_learning.test_cgp_pl_ale
python3 -m examples.policy_learning.test_tgp_pl
python3 -m examples.program_synthesis.test_cgp_ps
python3 -m examples.program_synthesis.test_tgp_ps
```

7 Example output

The following output has been taken from a run with tree-based GP that evolves a solution for the quartic polynomial problem (Koza-1): $f(x) = x^4 + x^3 + x^2 + x$ with an tolerance acceptance criterion of 10^{-6}. The ideal solution is obtained after 7830 fitness evaluations.

```
Generation #0 - Best Fitness: 4.549530199736617
Generation #1 - Best Fitness: 4.549530199736617
Generation #2 - Best Fitness: 4.549530199736617
Generation #3 - Best Fitness: 4.549530199736617
Generation #4 - Best Fitness: 4.549530199736617
Generation #5 - Best Fitness: 2.525883671205245
Generation #6 - Best Fitness: 2.525883671205245
Generation #7 - Best Fitness: 2.525883671205245
Generation #8 - Best Fitness: 2.525883671205245
Generation #9 - Best Fitness: 2.525883671205245
Generation #10 - Best Fitness: 2.525883671205245
Generation #11 - Best Fitness: 2.525883671205245
Generation #12 - Best Fitness: 2.525883671205245
Generation #13 - Best Fitness: 1.2524703496552547e-15
Job #0 - Evaluations: 7830 - Best Fitness: 1.2524703496552547e-15
```

8 Contributing

This repository is kept under a Github Organization to allow for a more inviting environment for contributions. The organization will not be tied to any specific institution and will be open to all contributors. If you want to contribute, please contact the maintainers to be added to the organization as a maintainer. The codebase is still in its early stages and contributions are welcome. If you have any suggestions, bug reports, or feature requests, please open an issue, submit a pull request or open a new discussion

8.1 Creating a new representation

To create a new representation, you can follow the following steps:

- Create a new Python script in the `src/gp` folder with the implementation of the representation. As a convention, name the script `tiny_<first letter of the representation>gp.py`. For example, `tiny_tgp.py` for Tree-based Genetic Programming and `tiny_cgp.py` for Cartesian Genetic Programming.
- Implement the representation as a class and create a `Tiny<first letter of the representation>GP` class that inherits from the `GPModel` class. The `GPModel` class is an abstract class that defines the interface for the different representations. This class should contain the following fields:
  - `config`: the configuration class inherited from Config abstract class.
  - `hyperparameters`: the hyperparameters class inherited from Hyperparameters abstract class.
  - `problem`: the problem class.
  - `functions`: a list of functions (non-terminals)
  - Implement the following methods in the `Tiny<first letter of the representation>GP` class:
    - `fitness(self, individual)`: the fitness function of a single individual.
    - `evolve(self)`: the evolution method that evolves the population.

    * `selection(self)`: the selection method that selects individuals for recombination and perturbation.
    * `predict(self, genome, observation)`: the prediction method that predicts the output of genome to a single observation.
    * textttexpression(self, genome): the expression method that returns the expression represented by genome.

8.2 Creating a new problem domain module

To create a new problem domain, you can follow the following steps:

- Update the file `src/gp/problem.py` with a new class that inherits from the Problem abstract class. This class should contain the following methods:
  - `is_ideal(self, fitness)`: a method that returns True if the fitness reached an ideal state (i.e., known optima).
  - `is_better(self, fitness1, fitness2)`: a method that returns True if fitness1 is better than fitness2.
  - `evaluate(self, genome, GPModel)`: a method that instructs how to evaluate a given genome using a GPModel

A good starting point is to look at the BlackBox and PolicySearch classes in the problem.py file which gives examples of two very different problem domains.
Finally, if you want to create an interface to an existing benchmark suite, you can look at the examples in:

- `src/benchmark/symbolic_regression/srbench.py`: interface to the SRBench benchmark suite.

9 Vision and Philosophy

TinyverseGP can be understood as a collection of minimalistic implementations of different representations for Genetic Programming. The goal is to provide a simple and easy-to-understand codebase with the following goals in mind:

- **Minimalistic**: The codebase should be as small as possible, while still being able to demonstrate the core concepts of the representation.
- **Extensible**: The codebase should be easy to extend and modify, so that it can be used as a basis for further research and experimentation.
- **Benchmarking**: The codebase should be able to run on standard benchmark problems and datasets, so that it can be used to compare different representations and algorithms.

10 Roadmap

Table 1 provides an overview of the current roadmap for TineverseGP. This table comprehends which problem domains and benchmarks are supported and which are planned in the framework of future extensions, the functionalities, and todo items. The checkmark with round bracket states that the problem domain can be approached with the current version of TinverseGP but the full support of corresponding benchmarks is still missing.

Table 1: Currently planned support for different benchmars and problem domains, ✓means it is already supported, (✓) means it has partial support, × means it is under development. More benchmarks will be considered for the future versions.

Problem Domains	**Support**	**Benchmarks**
Logic Synthesis	✓	Classical, GBFS
Symbolic Regression	(✓)	Classical, SRBench, Feynman
Policy Search	(✓)	Gymnasium (CartPole, LunarLander)
Program Synthesis	×	Leet Code, PBS1, PBS2, SyGuS
ODEs and PDEs	×	ODEBench
Functionalities	**Support**	**Notes**
Autodiff	×	own implementation or a library like JAX
SMT solver	×	integration with Z3
Numerical constants optimization	×	NLOpt
Program compilation	×	
Multithreading	×	
TODO	**Support**	**Notes**
Refactor *GPModel*	(✓)	Implement defaults for *evolve* function