Search Loop and Step

Search Loop

The search loop in PonyGE2 controls the overall duration of the search process, including the initialisation of the initial population, along with the main generations loop. In canonical GE, the search process loops over the total number of specified generations.

NOTE *that in PonyGE2 the total number of generations refers to the number of generations over which the search process loops (i.e. over which evolution occurs),* ***NOT*** *including initialisation. Thus, specifying 50 generations will mean an initial population will be generated and evaluated, and then the evolutionary process will loop for 50 generations. Since the initialised generation will be Generation 0, the total number of individuals evaluated across an entire evolutionary run will by* ***population x (generations + 1)****.*

The default search loop function in PonyGE2 is stored in `algorithm.search_loop`. However, users are free to implement their own search loops. These search loops can be stored either in the default `algorithm.search_loop` module, or wherever users desire. The search loop is a parameterisable function that can be [[set in the parameters dictionary|Evolutionary-Parameters#setting-parameters]].

While PonyGE2 is currently set up to only use the main search loop function (save for special cases such as re-loading an evolutionary run from state), it is possible for users to write their own search loop functions. It is possible to specify the desired search loop function directly through the parameters dictionary with the argument:

```
--search_loop [SEARCH_LOOP]
```

or by setting the parameter `SEARCH_LOOP` to `[SEARCH_LOOP]` in either a parameters file or in the params dictionary, where `[SEARCH_LOOP]` is the name of the desired search loop function.

Along with the default search loop, a number of [[Hill Climbing|Search-Options#hill-climbing]] search loops are provided in PonyGE2.

Step

At each generation in the main search loop, the `step` function is called, which by default is `algorithm.step.step()`. The `step` function executes one full iteration of the canonical evolutionary process, typically:

1. [[Selection]]
2. [[Variation]]
    - [[Crossover|Variation#crossover]]
    - [[Mutation|Variation#mutation]]
3. [[Evaluation]]
4. [[Replacement]]

Step functions are stored in `algorithm.step`. While PonyGE2 is currently set up to only use the main step function, it is possible for users to write their own step functions. It is possible to specify the desired step function directly through the parameters dictionary with the argument:

```
--step [STEP]
```

or by or by setting the parameter `STEP` to `[STEP]` in either a parameters file or in the params dictionary, where `[STEP]` is the name of the desired step function.

***NOTE** that if a step function is saved in the main `algorithm.step` module, there is no need to specify the full path to the function. However, if the desired step function is located elsewhere, it **is** necessary to specify the full module path to the function.*

Population Options

There are a number of parameters within PonyGE2 for controlling overall populations.

Population Size

The population size controls the total number of individuals to be generated at each generation. The default value is 500. This value can be changed with the argument:

```
--population_size [INT]
```

or by setting the parameter `POPULATION_SIZE` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the population size.

Higher population sizes can improve performance on difficult problems, but require more computational effort and may lead to premature convergence.

Generations

The number of generations the evolutionary algorithm will run for. The default value is 50. This value can be changed with the argument:

```
--generations [INT]
```

or by setting the parameter `GENERATIONS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of generations.

Higher numbers of generations can improve performance, but will lead to longer run-times.

Hill Climbing

A simple hill-climbing approach, and two modern variants, are provided in `algorithm.hill_climbing`. In simple HC, there is a single current individual (i.e. no population). At each step of the search loop, a mutation operator creates a new candidate individual. If this individual is better than the original, the move is "accepted" -- the candidate replaces the current individual. In Late-Acceptance Hill-Climbing (LAHC), due to Bykov, and Step-Counting Hill-Climbing (SCHC), due to Bykov, the acceptance decision depends on whether the candidate is better than one considered some time previously, dependent on a "history length" parameter. This can allow search to escape local optima. The following arguments will select one of these methods:

```
--search_loop algorithm.hill_climbing.LAHC_search_loop

--search_loop algorithm.hill_climbing.SCHC_search_loop
```

or by setting the parameter `SEARCH_LOOP` to either `algorithm.hill_climbing.LAHC_search_loop` or `algorithm.hill_climbing.SCHC_search_loop` in either a parameters file or the parameters dictionary.

The "history length" parameter is set as follows:

```
--hill_climbing_history [INT]
```

or by setting the parameter `HILL_CLIMBING_HISTORY` to `[INT]` in either a parameters file or the parameters dictionary, where `[INT]` is an integer value specifying the desired hill climbing history length.

Step-Counting Hill Climbing has an additional `SCHC_COUNT_METHOD` parameter that can be specified. This parameter sets the counting method to be used with the SCHC algorithm. This parameter can be set with the argument:

```
--schc_count_method [METHOD]
```

or by setting the parameter `SCHC_COUNT_METHOD` to `[METHOD]` in either a parameters file or the parameters dictionary, where `[METHOD]` is a string specifying the desired step counting method. Currently implemented acceptable strings are:

1. "count_all"

   This is the default option, and counts all moves.

2. "acp"

   This option counts accepted moves only (i.e. where the candidate solution is better than the cost bound and better than or equal to the current best).

3. "imp"

> This option counts improving moves only (i.e. where the candidate solution is better than the current best).

***NOTE** that both LAHC and SCHC reduce to a simple hill-climbing algorithm when used with a history length of 1.*

***NOTE** that crossover is **not** used in hill-climbing algorithms.*

Distributed Search

In the standard implementation of PonyGE2, the evolutionary search process runs on a single machine (or agent) which has access to all the individuals in the population. It can then choose among the highest performing individuals from the population to perform the various genetic operations. In a multi-agent implementation, each agent has its own instance of genetic information. An agent can only have access to its own genotype. The agent moves in an environment and communicates to nearby agents to share the genetic information. After collecting genetic information from a handful of agents, it can apply the genetic operations by itself for the information that it has gathered. This multi-agent approach is particularly useful for robotics or distributed implementations across many nodes, as each agent can act individually with only local knowledge.

Distributed Search Loop

When the `MULTIAGENT` parameter is set to `True`, search_loop and step methods from `algorithm.distributed_algorithm` automatically get activated. The first step in the distributed search loop is to initialize the various agents objects. Based on the parameter set for `AGENT_SIZE` that the number of agents is created, where the `INTERACTION_PROBABILITY` parameter is passed while initializing agents. This parameter defines the probability of interaction with other agents. Higher this value, there are higher changes interacting with other agents and higher chances of finding solutions faster. Then after initializing agents, for each generation, step module is called which takes the list of agents as a function parameter and returns a list of agents.

Distributed Step

In this module, for each agent in the environment, the agent will perform: sense, act and update action sequentially. These three action or modules are built in for the agents. These modules can be extended or overridden based on the type of agent required to solve a particular problem. After performing these actions, a list of agents is returned.

Agents

It is the class that defines the behavior of the agents. Right now we have only defined a very primitive agent but based on the requirements the method of this class can be overridden. The Sense method describes the way the agent gets information from the environment. The Act method provides a way from the agent to process the information obtained from Sense method. The update method is agent's way to update its state. In our case, the Sense method is used to find other nearby agents, the Act method is used to apply genetic operations to the stack of genetic information collected from nearby agents and the update method is used to update the agent's genetic information.