Project III: Biogimmickry

Due Wed, Jul 21 by 9PM Pacific Time

To begin, use the following file: http://users.csc.calpoly.edu/~dkauffma/480/biogimmickry.py

- To download locally, right-click link and choose "Save link as...".
- To download and unzip on a CSL server run:

```
wget http://users.csc.calpoly.edu/~dkauffma/480/biogimmickry.py
unzip biogimmickry.py
```

Description

In this project, you will write a program that itself writes programs. These programs will be run using a simple interpreter to write integers to a memory - represented as a fixed-length array - using a minimalist language that supports the commands listed below. This language derives from Brainfuck (BF), an esoteric programming language designed by Urban Müller.

Command	Description
>	Increment memory pointer
<	Decrement memory pointer
+	Increment integer at current memory location
-	Decrement integer at current memory location
[	If cell at memory pointer is zero, jump to corresponding `]` command Otherwise, move to next command
]	If cell at memory pointer is non-zero, jump to corresponding `[` command Otherwise move to next command

Thus, the programs generated will be sequences of these commands and, when interpreted, must write values to an empty memory that exactly matches a provided target array. To create these programs, you will implement a genetic algorithm, a local search method loosely based off of biological evolution for finding sequences of values that meet given criteria.

For a demonstration of the language, see this Brainfuck Visualizer.

Genetic Algorithms

Local search algorithms often employ a strategy of creating many candidate solutions to a problem and exploring the states around the best ones. While this approach can be effective for simple problems, global optima may not be found unless the number of states being explored is intractably high. Genetic algorithms (or GAs) improve on these algorithms by using more sophisticated methods for selecting individual states and using their components to create more diverse populations in subsequent generations.

Fitness Functions

An objective (or evaluation) function is necessary to assess the quality of each individual in a generation; in GA parlance, this method is referred to as a fitness function. Unlike the other components of a GA, the implementation of a fitness function is specific to the problem domain and often represents its most costly operation.

For this project, a natural fitness evaluation would compare the contents of the array modified by an individual program against a target array that the program is supposed to produce. This comparison can be made by calculating the sum of the absolute difference between the two arrays. Additional metrics may also be used, such as attributing higher scores to smaller programs to promote brevity; however, while such additions may be used to break ties, accuracy should be the primary basis for evaluation.

Selection

Once all of the individuals in a generation have been scored, the next generation can be constructed. To do so, individuals from the current generation must be selected (typically in pairs) for the crossover stage. Prior to selection, the entire generation's fitness scores are usually normalized to probabilities (if lower fitness scores indicate better individuals, simply subtract each score from the sum of all scores). Afterward, individuals may be randomly sampled from the population and their corresponding weights accumulated until this sum exceeds a randomized threshold, at which point that individual is selected.

In addition to this approach, it can also be beneficial to only consider for selection individuals that fall within a certain top percentile. However, it is important not to remove too many individuals from consideration or else the population will converge too quickly.

Crossover and Mutation

Once two individuals have been selected, the crossover method specifies at which point the individuals will be divided and recombined to create new individuals. Typically, two-point crossover is used in which both individuals are divided at the same point and two successor individuals result from swapping either of the two segments. Other forms of crossover exist, including three-point and uniform crossover, the latter of which swaps individual elements of the sequents instead of segments as a whole.

In addition, GAs often provide a chance for the resulting successor individuals to undergo mutation, whereby one or more of their elements are randomly changed. This step can help prevent individuals from converging by promoting diversity; however, if used too frequently, the benefits of crossover are lost and the search becomes a random walk. Mutation may include different probabilities for adding, deleting, or modifying parts of an individual.

Implementation

Allowed Modules: `random`

Your GA implementation should not deviate too far from methods discussed in lecture. You may need to experiment with different parameters to find a balance that works in a timely manner. These parameters include (not exhaustively):

- Size of the starting population
- Lower and upper bounds on program lengths

- Percentile kept for the next generation
- Mutation rates

The `biogimmickry.py` file (linked above) provides a `FitnessEvaluator` class that performs scoring for a program, with a score of zero indicating that the program, when interpreted, populates a memory array that exactly matches an array that was used to instantiate the `FitnessEvaluator`. It is this score that must be used for the selection process, as well as determining whether the search is complete.

The `FitnessEvaluator` class should not be modified. It is instantiated once prior to a search and is accessed using `FitnessEvaluator.evaluate(program)` to receive a score for a `program` string. For an example of how to instantiate it, see the `main` function in the file.

Required Functions

In the following instructions, "program" refers to a BF program, and "loop" refers to loops in BF programs (not Python loops). The "target array" is what the BF interpreter's memory must match when a solution program is run.

Assume all target arrays have an **absolute sum** no greater than `20` (e.g. `(2, -4, 6, -8)`).

```
create_program(fe: FitnessEvaluator,
max_len: int) -> str
```

Return a program string no longer than `max_len` that, when interpreted, populates a memory array that exactly matches a target array.

The `max_len` argument forces programs to use loops by limiting their length. If `max_len` is zero, it may be disregarded; in other words, a loop will not be necessary to match the target array. In this case, you may choose an appropriate upper bound for program lengths, keeping in mind that longer programs take more time to evaluate. However, if `max_len` is positive, the length of the program returned by this function must not exceed this value. Use this distinction to first ensure your algorithm correctly generates programs without loops before adding the logic to match arrays iteratively.

For non-loop programs, the target array will never exceed a length of `8`; for programs requiring a loop, the target array length will always be `2`, with one element set to zero (to be used as a loop counter) and the absolute value of the other element greater than `max_len`. For example, a target array might be `(0, 20)` with a `max_len` of `18`. Thus, because the length of the program must be less than the absolute sum of the array, a proper use of the loop (with a counter) must be found by the genetic algorithm to match the target array.

Note that the minimum length of any program containing a loop should be `12`, as a loop in a shorter program would not be necessary or meaningful. For example, a minimum-length loop-containing program might look like `"+++[>++++<-]"`. Programs without loops may be of any length.

When implementing this function, you may want to follow these steps:

1. Initialize a population of random programs
   - Population size should be as large as possible (usually over `500`) without significantly slowing down each generation
   - Programs should never be greater in length than `max_len` (if non-zero)
   - Do not create programs with more than one loop; that way lies madness
2. Score all programs using `FitnessEvaluator.evaluate(program)`
   - Stop if a program has a fitness score of zero
3. Select two programs in the top `N` percentile weighted by score using `random.choices` (or some other weighting method)
   a. Use crossover to create two new programs, ensuring both do not exceed `max_len` (if non-zero)
      If a loop exists, avoid breaking it by ensuring the crossover point is:
      - To the left of both programs' `[` command
      - To the right of both programs' `]` command
      - Between both programs' `[` and `]` commands
   b. Randomly mutate zero or more elements in each of the new programs

   - Mutation may be any of addition, deletion, or editing of commands, each with separate probabilities
   - Ensure that mutation maintains one loop in the program and does not cause it to exceed `max_len`

  c. Add the new programs to the next generation

4. Repeat Step 3 until the next generation is the size of the previous generation
5. Go to Step 2 using the next generation UNLESS it has converged, in which case go to Step 1

Scoring Rubric

The score you receive on this assignment will be based on which achievements the program satisfies. The achievements are listed in the order recommended to complete them.

	Achievement	Credit
1	Matches 1 Length Array	15%
2	Matches 2-4 Length Non-Negative Array	20%
3	Matches 5-8 Length Array	30%
4	Iteratively Matches 2 Length Array	35%

Submission

On a CSL server with `biogimmickry.py` in your current directory:

Instructor	Command
Daniel Kauffman	`/home/dkauffma/casey 480 biogimmickry`