Natural Computing Assignment 2019/2020

Genetic Programming

Exam Number: B145418

Contents

1 Genetic Program Design Report

1.1 Main Ideas and Approach

I used an object-oriented approach with Python programming language to design and implement a genetic program (GP) that evolves a fixed-sized population of individuals (representing functions) over several generations using crossover, mutation and selection as evolutionary drivers. The goal of the GP was to find formulas that produce many prime numbers. In addition to designing my own Python classes to implement the GP, I also used an existing programming framework, Distributed Evolutionary Algorithms in Python (DEAP)[1], to implement "function tree" data structures that were useful in representing the solutions found by the GP.

Fitness of individuals was measured as the number of unique prime numbers obtained in the range set given a domain set consisting of all positive integers up to K, consequently setting up a fitness maximization problem. The fitness objective was chosen because functions that maximize this fitness objective also produce many prime numbers as per the goal of this assignment. Moreover, the fitness objective chosen is mathematically precise and is easily implemented using a combination of a primality checker and counting operation.

1.2 Technical Details and Implementation

The technical details and implementation of attributes of the GP are described in Table 1 below.

Table 1: Technical Details and Implementation

Attribute	Description
Individual & Fitness - *individual.py* - *custom_operator.py* - *primitive.py*	Implemented an Individual class that has a "function tree" and a fitness value as key attributes. Additionally, instances of the Individual class (individuals) contain utility methods to perform actions such as calculating fitness, visualizing the tree structure of the individual etc. The "function tree" is made up of nodes selected from a DEAP primitive set that includes terminals (i.e. constant 'a' and argument k) and non-terminals (i.e. multiplication and addition operators). At initialization, the constant 'a' is initialized as a random prime number. Calculated fitness by compiling the function represented by the "function tree", and then counting all the unique prime numbers identified in the range of the function, given a specific domain.
Primality Checking - *primality_checker.py*	I implemented a class (primality checker) class that has a prime number test method in which a primality checker object is initialized with a list of the first 1229 prime numbers; numbers are checked against this list to determine primality. Admittedly, the custom primality checker is limited to only the prime numbers that are less than 9973, however, this set of prime numbers is deemed sufficient for prime numbers generated using values of K < 100 (since 99^2=9801 which is less than 9973). As a result, the fitness used by the GP is valid only for values of K < 100, and fitness ranges from 0 to 100.
Population - *population.py*	Implemented a Population class that has a dictionary of individuals as its key attribute. Only a single population is instantiated for during each run of the GP. The size of the population is kept constant across generations by dropping the least fit individuals.
Cross Over - *evolution.py*	Cross over was implemented by swapping nodes between two randomly selected individuals as follows: - randomly pair up individuals within the population (parents) - make copies of the paired individuals (children) - randomly select a node in one of the children and swap it with a randomly selected node of the same kind (terminal or non-terminal) in child. - For each child, If the child is fitter than its parent then add the child to the population, if the child is equally as fit as the parent then replace the parent with the child, otherwise drop the child.
Mutation - *evolution.py*	Mutation was implemented by randomly selecting a node and replacing with a different node of the same kind (i.e. terminal or non-terminal)
Selection - *evolution.py*	Selection was implemented by dropping the least fit members in order to maintain the population size across generations.
Inputs - *controller.py* - *experiment_list.xlsx*	Multiple runs of the GP can be executed by listing the input parameters for each run in the spreadsheet file *experiment_list.xlsx.* The input parameters are read into the program by the controller.
Outputs - *experiement-id.txt*	Each run of the GP generates a text file summarizing the input parameters used for the run, and a list the resulting population of individuals with the fitness score of each individual.

[1] DEAP Documentation https://deap.readthedocs.io/en/master/ (accessed on: November 9, 2019)

1.3 Experiments Conducted

I conducted seven experiments (runs) of the GP. The input parameters used for each experiment are summarized in Table 2 below. The experiments aimed to roughly examine the impact of changing following parameters on the performance of the GP:

1. The maximum number of allowed generations
2. The population size
3. Mutation probability

The GP's performance was assessed by comparing the run time, and the fitness of the fittest individual in the evolved population, and the number of "good" solutions discovered by the GP. Solutions with fitness scores greater than or equal to 50 (i.e. generated primes at least half of the time) were considered to be "good" solutions.

Table 2: Summary of Experiments Conducted

Experiment Id	Maximum Number of Generations	Early stopping threshold[2]	Population size	K max[3]	Minimum tree depth	Maximum tree depth	a range[4]	Mutation probability
001	100	0.8	100	100	2	6	1,10	0.3
002	500	0.8	100	100	2	6	1,10	0.3
003	1000	0.8	100	100	2	6	1,10	0.3
004	100	0.8	500	100	2	6	1,10	0.3
005	100	0.8	1000	100	2	6	1,10	0.3
006	100	0.8	1000	100	2	6	1,10	0.1
007	100	0.8	1000	100	2	6	1,10	0.5

1.4 Results

Table 3 summarizes the results of each experiment[5].

Table 3: Summary of Results

Experiment Id	Run Time (s)	Best Solution [fitness]	Other Good Solutions [fitness]
001	20.5	$k^2 + k + 19$ [60]	none
002	21.9	$k^2 + k + 41$ [86]	$6k + 41$ [53] $12k + 41$ [50]
003	13.5	$k^2 + k + 41$ [86]	$6k + 41$ [53] $12k + 41$ [50]
004	82.3	$k^2 + k + 41$ [86]	$k^2 + k + 17$ [60] $6k + 5$ [57] $k^2 + 9k + 37$ [56] $6k + 13$ [51] $12k + 7$ [50] $12k + 17$ [50]
005	35.8	$k^2 + k + 41$ [86]	none
006	78.8	$k^2 + k + 41$ [86]	$6k - 19$ [54]
007	75.7	$k^2 + k + 41$ [86]	$k^2 + k + 17$ [60] $6k + 17$ [55] $6k + 41$ [53] $k^2 + k + 67$ [51]

[2] The GP terminates either when the maximum number of iterations is reached or when there exists an individual in the population with fitness value that exceeds 80% of K max.

[3] K max refers to the size of the domain set that is used to evaluate the fitness of individuals

[4] Constant 'a' is a seed prime number that is used by the GP. For each individual, the value of a is initialized by randomly selecting a prime number from within the specified range (a range).

[5] Raw experimental result data for each experiment is in the associated text file. The sorted results are in the spreadsheet *sorted_Results.xlsx*.

As shown in Table 3, the best overall solution discovered by the GP is one of the Euler formulas $k^2 + k + 41$, which generates 86 prime numbers given the defined domain set. Overall, the GP finds the best result reliably (I.e. in 6 out of 7 runs). Moreover, the GP is also able to find other relatively good solutions, including some that are not of the form $k^2 \pm ak \pm b$. The best solution generates 43% more unique prime numbers given the domain set, compared to the second-best solution.

The size of the program is controlled by imposing restrictions on the population size, "function tree" depth, the size of the domain set (K max) and the maximum number of generations/iterations.

Looking at experiments 1, 2 and 3 where the number of allowed generations is increased while fixing all other parameters, it is evident that the GP is able to find additional good solutions in later generations. However, based on the relative run times of experiments 1, 2 and 3, it is likely that the GP invoked early termination in experiment 3 after discovering the Euler formula with a fitness value (86) that exceeds the early termination threshold (80).

Looking at experiments 1, 4 and 5 where the population size was increased while fixing all other parameters, it is evident that a population size of 500 is more optimal than a population size of 100, based on the set of good solutions generated in each experiment. For experiment 1, it is likely that a population of 100 does not have enough diversity to yield multiple good solutions. Based on the relative run times for experiments 4 and 5, it is likely that the GP invoked early termination in experiment 5 after discovering the Euler formula with a fitness value (86) that exceeds the early termination threshold (80).

Looking at experiment 6 and 7, it is evident that increasing the probability of mutation yielded a larger set of good solutions.

1.5 Alternatives Considered

As an alternative to using a custom-built primality checker, I considered using MATLAB's prime number checking function (`isprime`) to implement primality checking as part of fitness evaluation. This was implemented by importing MATLAB's computation engine[6] in Python to dynamically launch a MATLAB process at run time, as opposed to populating a finite list of prime numbers at program initialization. The MATLAB approach would remove the limitation on the size of the domain set for which the GP produces valid solutions. However, in addition to increasing the GP's overall run time, this approach introduces a dependency on a proprietary software, MATLAB. Moreover, none of the two approaches consistently yields superior solutions for values of k_max < 100. Nonetheless, I implemented the approach of using MATLAB's `isprime` function as a non-default option that can be selected by the program user if deemed necessary.

When developing the GP, I also considered adding a subtraction operator to the primitive set. However, preliminary runs of the GP demonstrated that the subtraction operator was redundant since the selected mutation method allows the GP to explore negative values of the constant 'a', without necessitating an explicit subtraction operation.

The DEAP framework includes generic methods that can be used for implementing crossover, mutation and selection. However, in order to gain deeper insight into the behavior of the GP, I elected to implement my own methods for cross-over, mutation and selection.

1.6 Solution Applicability and Generalization

The GP discussed here could work well for a time series prediction application if the argument k is considered to represent time. However, a new fitness objective would have to be defined based on a training data set that specifies the desirable outcome for different values of k. For example, fitness could be calculated as the number of inputs k for which an individual's predictions are desirable. A population of individuals would then be evolved, potentially finding a good predictor.

[6] https://www.mathworks.com/help/matlab/matlab_external/call-matlab-functions-from-python.html