Invasive Weed algorithm

Xoel Mato Blanco

18/02/2020

1. Introduction

Invasive weed optimization algorithms are based on the idea of plant's growth behaviour as a process that will constantly create new plants unless some restriction is presented, such as the maximum number of plants. The quantity of new seeds that a plant contributes to the new generation is also adjusted to maximise that number for the best plants in the population. How good a plant is is assessed using a fitness function. Since we are trying to find the minimum of the function, this acts as a "cost" function.
Building up on those principles, I coded the algorithm as presented in the python script **InvasiveWeed.py**.

2. Code

First, all the modules I'll need for this model:

```
import math
import time
import numpy as np
import random
import matplotlib.pyplot as plt
```

2.1. Invasive weed optimization model

The function to minimize is the one shown below:

```
def eval_fitness(x, y):
    return x * math.sin(4 * x) + 1.1 * y * math.sin(2 * y)
```

The class defined below contains the model and the functions required to initialize it, iterate to minimize the eval_function and store the necessary results:

```
# This class stores the model.
class InvWeed:
    # This function is the main function of the model. It executes all the
    # required calls to the rest of the functions.
    def __init__(self, initial_size=5, pmax=200, new_seeds=100, niter=100, delta=1e-6,
                 max_rep=10, verbose=False):
        # Initializes a counter for the runtime, counts the number of iterations
        # and save the parameters given.
        self.runtime = time.time()
        self.parameters = [initial_size, pmax, new_seeds, niter, delta, max_rep]
        self.niters = 0
        # Initializes the population using a random value in the interval proposed.
        # The population is a list of lists, where each element is a
        # different plant [x, y, fitness].
        self.pop = []
        for i in range(initial_size):
            x = random.uniform(0, 10)
            y = random.uniform(0, 10)
```

```
            new_plant = [x, y, eval_fitness(x, y)]
            self.pop.append(new_plant)
        # This line keeps the population sorted according to their fitnesses.
        self.pop.sort(key=lambda x: x[2])
        # On records I maintain a list with the best fitness of each generation.
        self.records = [self.pop[0][2]]
        # This will print the summary of this generation.
        if verbose:
            self.summary_print(self.niters)
        # The population will evolve using the next command.
        self.iterate(pmax, new_seeds, niter, delta, max_rep, verbose)
        # This saves the overall best plant, only its x and y values.
        self.best_plant = [self.pop[0][0], self.pop[0][1]]
        # Saves the runtime.
        self.runtime = time.time() - self.runtime

    # Iterate will make the population evolve until certain conditions are met.
    def iterate(self, pmax, new_seeds, niter, delta, max_rep, v):
        # This counter keeps how many the fitness has not improved.
        counter = 0
        # The maximum number of iterations is given by niter.
        for self.niters in range(1, niter):
            # Reproduce generates new plants in the population.
            self.reproduce(new_seeds)
            # If there's more plants than allowed, the worst of them are discarded.
            if len(self.pop) > pmax:
                self.pop = self.pop[:pmax]
            # Saves the new best plant.
            self.records.append(self.pop[0][2])
            # Prints the summary of the generation.
            if v:
                self.summary_print(self.niters)
            # If the fitness has not improved, it increases the counter and breaks if it
            # happened too many consecutive times.
            if self.records[self.niters - 1] - self.records[self.niters] < delta:
                counter += 1
                if counter > max_rep:
                    break
            # If not, the counter is reset.
            else:
                counter = 0

    # Given a population and the number of new seeds to generate, it will create those
    # new individuals.
    def reproduce(self, seeds_gen):
        # I transform the fitnesses to be in a scale that is proportional
        # to the initial one. All fitnesses in there are positive and the maximum value
        # corresponds to the minimum original value.
        all_fitness = [x[2] * -1 + self.pop[-1][2] for x in self.pop]
        # Here I build an iterator to have consecutive values in a range to use later as
        # standard deviation.
        sd_iterator = iter(list(np.linspace(0.4, 2, seeds_gen)))
        # In this line, several things happen: I get as many random elements from the
```

```
        # population as given, but the probability of choosing any of them is
        # proportional to their fitness and the best fitness in the population. This
        # list of individuals is sorted so that the first one will be the one with
        # the best fitness.
        for plant in sorted(random.choices(self.pop,
                                           weights=[x / max(all_fitness)
                                                    for x in all_fitness],
                                           k=seeds_gen),
                            key=lambda x: x[2]):
            # Each of the new seeds is created in this function and added to the
            # population.
            # The best individuals get the lowest standard deviation values.
            self.pop.append(self.disperse(plant[0], plant[1], next(sd_iterator)))
        # This keeps the population sorted.
        self.pop.sort(key=lambda x: x[2])

    # Disperse takes a data point (x, y) and a standard deviation and returns a new plant.
    def disperse(self, x, y, omega):
        # The new data points are created from a gaussian distribution N(x or y, s.d.)
        new_x = random.gauss(x, omega)
        new_y = random.gauss(y, omega)
        # The while loops check that the values are in the range established.
        while new_x <= 0 or new_x >= 10:
            new_x = random.gauss(x, omega)
        while new_y <= 0 or new_y >= 10:
            new_y = random.gauss(y, omega)
        # This will return the new plant, with x, y and fitness values.
        return [new_x, new_y, eval_fitness(new_x, new_y)]

    # This will print some useful information on the current generation.
    def summary_print(self, n):
        print("Generation:", n, '\n\tPopulation size', len(self.pop), '\n\tBest plant',
              self.pop[0], "\n")

    # Plot will show the results of the current model.
    def plot(self):
        plt.plot(self.records)
        plt.ylabel('Cost function')
        plt.xlabel('# Generation')
        plt.title(str(self.parameters))
        plt.show()
```

2.1.1. Notes on the Invasive Weed model

- The way I transformed the fitness results to the number of new seeds by that plant is the best one I found, but I see how this is a point to be improved and tested in more detail.
- The way that I chose the values for the new seeds from their parent plant was a gaussian distribution. Other distibutions and methods could be tried, but I didn't focus on that point.
- Another part of the code to check is how the not in range values are transformed to valid values.
- The standard deviation range did not have an important impact on the results I checked.

2.2. Tuning the hyperparameters: grid search

I wrote the previous code as parameter-dependent as I could. This allowed me to build a new class to look for the best hyperparameters of my implementation:

```
# Import for multiprocessing
import multiprocessing

# Declaration of the class.
class GridSearch:
    # Initializing the class.
    def __init__(self, function, pars, reps=3):
        # Prints information on the number of combinations possible (h)
        print('Looking for best parameters:')
        h = 1
        for x in pars.values():
            h *= len(x)
        print('\t· ' + str(h), 'combinations being tested.\n\t· Results averaged using',
              reps, 'repetitions.')
        # Get combinations creates a list of lists with all the parameter combinations
        self.test_list = self.get_combinations(pars, reps)
        # Initializes a multiprocessing object
        pool = multiprocessing.Pool(processes=8)
        # Loads the models for all the parameters parsed using multiprocessing to speed up
        # computations
        self.models = [pool.apply_async(function,args=(x[0], x[1], x[2], x[3], x[4], x[5],
                                                        )) for x in self.test_list]
        # All the resulting models are stored in results using another function
        self.results = self.extract_results(self.models)
        # Stores the best fitness and runtimes
        self.best_fitness = min([x[0] for x in self.results])
        self.best_runtime = min([x[1] for x in self.results])
        # This computes how good the model is
        self.compute_distance()
        # This writes the results to a txt file´
        self.write_results()
        # This saves the best parameters
        self.best_pars = self.results[0][-2]

    # Get combinations creates a list iterating over all the possibilities in the
    # parameters dictionary
    def get_combinations(self, pars, reps):
        par_list = []
        for pi in pars['initial_size']:
            for pmax in pars['pmax']:
                for new_seeds in pars['new_seeds']:
                    for niter in pars['niter']:
                        for delta in pars['delta']:
                            for max_rep in pars['max_rep']:
                                # It takes into account how many times the results have
                                # to be estimated
                                for _ in range(reps):
                                    par_list.append([pi, pmax, new_seeds, niter, delta,
                                                     max_rep])
```

```
        return par_list

    # Extract results from the fitted model list
    def extract_results(self, models):
        # results will store some results from the fitted models
        results = []
        # dict_help will store the results of different parameter list as key
        dict_help = {}
        # Iterates on the models
        for p in models:
            # If this parameter list has not been seen yet:
            if not tuple(p.get().parameters) in dict_help.keys():
                # Saves the results as value of those parameters as key
                dict_help[tuple(p.get().parameters)] = [[p.get().records[-1],
                                                        p.get().runtime,
                                                        p.get().niters]]
            # If the parameters have already results stored, it will append the new ones
            else:
                dict_help[tuple(p.get().parameters)].append([p.get().records[-1],
                                                             p.get().runtime,
                                                             p.get().niters])
        # For each parameter list in the dictionary, it will average the results stored
        for k in dict_help.keys():
            rec = 0
            runt = 0
            nit = 0
            # The results are estimated from the results in the dictionary
            for r in dict_help[k]:
                rec += r[0] / len(dict_help[k])
                runt += r[1] / len(dict_help[k])
                nit += r[2] / len(dict_help[k])
            # And they are appended to the results list
            results.append([rec, runt, nit, list(k)])
        return results

    # Compute distance will give a score to each model.
    def compute_distance(self):
        for c in self.results:
            # The score is computed moving the point in space according to the best
            # values in the list and
            # check its euclidean distance as a scoring measure.
            # This is appended to each of the results
            c.append(math.sqrt((c[0] - self.best_fitness) ** 2 +
                               (c[1] -self.best_runtime) ** 2))
        # The results are sorted using this new scoring method
        self.results.sort(key=lambda x: x[-1])

    # Opens the file and writes every result as a new line
    def write_results(self):
        with open('results.txt', 'wt') as file:
            for c in self.results:
                file.write(str(c))
                file.write("\n")
```

```
    # This plots how all the models compare, showing the runtimes and fitnesses
    def plot(self):
        plt.scatter([x[1] for x in self.results], [x[0] for x in self.results])
        plt.xlabel('Runtime (s)')
        plt.ylabel('Fitness value')
        plt.title('Grid search results')
        plt.show()
```

2.2.1. Notes on GridSearch

This grid search implementation would take quite long for an intensive search. To minimize the computing time, I implemented the eval_function in C and provided in the files. In the same way, running this with parallelization helped reducing this runtime.
2.3 Cython implementation of the eval_fitness function As I said before, I implemented the fitness function as a cython function in a separate pyx file. This allows c-like python code to be translated into proper C code. Thereby, the performance of the code was two times better while testing. Cythonising the rest of the code was not as trivial and presented some problems that I was not able to overcome. Nevertheless, cythonising only this function resulted in a good enough (2x) speed-up.

```
import cython
from libc.math cimport sin

cpdef float eval_fitness( float x, float y ):
    return x * sin(4 * x) + 1.1 * y * sin(2 * y)
```

Using cython requires the extra step of translating the code to a c file and then compile it to machine code with the following bash command. The resulting files have been attached so this step can be skipped.

```
cythonize -3 -i -a fitness_func.pyx
```

After compilation, the function can be imported as any other python module in the main script.

```
from fitness_func import eval_fitness
```

2.4 Main function

I finally wrote a main function to run all the necessary code:

```
# Declaration of the function.
def main(mode='short', nrep=3):
    # Setting up a timer
    t0 = time.time()
    # Checking which parameters dictionary to use.
    # The larger the parameters dictionary, the more accurate the parameters found will be.
    if mode == 'short':
        parameters = {
            'initial_size': [5, 100],
            'pmax': [200, 500, 1000],
            'new_seeds': [20, 100],
            'niter': [100, 1000],
            'delta': [1e-6, 1e-9],
            'max_rep': [1, 10]
        }
```

```
    else:
        parameters = {
            'initial_size': [5, 10, 100, 1000],
            'pmax': [100, 200, 500, 1000],
            'new_seeds': [20, 100, 200],
            'niter': [20, 100, 500],
            'delta': [1e-6, 1e-9],
            'max_rep': [5, 10, 20]
        }
    # Calling the GridSearch using the InvWeed class, the parameters dictionary and
    # the number of tests to run per combination
    search = GridSearch(InvWeed, parameters, nrep)
    # Plot the GridSeaerch results
    search.plot()
    # Get the best parameters found and print them
    pars = search.best_pars
    print('Best parameters found:', pars, '\n\nBuilding model...')
    # Rebuilding a model with those parameters and plot the evolution of the model
    model = InvWeed(pars[0], pars[1], pars[2], pars[3], pars[4], pars[5], True)
    model.plot()
    # Stop timer
    print(time.time() - t0)
```

3. Results

The results were similar most of the times I run this script. The best parameters found by GridSearch usually consisted of:
- Initial population size: 5 or 100
- Maximum population size: 200
- Number of new seeds per generation: 100
- Maximum number of iterations: 100
- Mimimum difference between generations (delta): 1e-6
- Maximum number of consecutive generations that the delta condition can be unmet: 10

Even though some of this parameters are quite stable, the last two parameters in this list were the ones that changed the most as I could observe, so their impact might not be that important.
One last consideration I would like to make is that my GridSearch implementation selects the best model as the one that achieves the best fitness in the shortest amount of time. However, the model that we tried to get is one which improvement curve is negative exponential rather than a semi-curve with plenty of flat regions. The problem is that I could not code a function that solved this. The results, though, are good enough.

This is how the results would be presented:

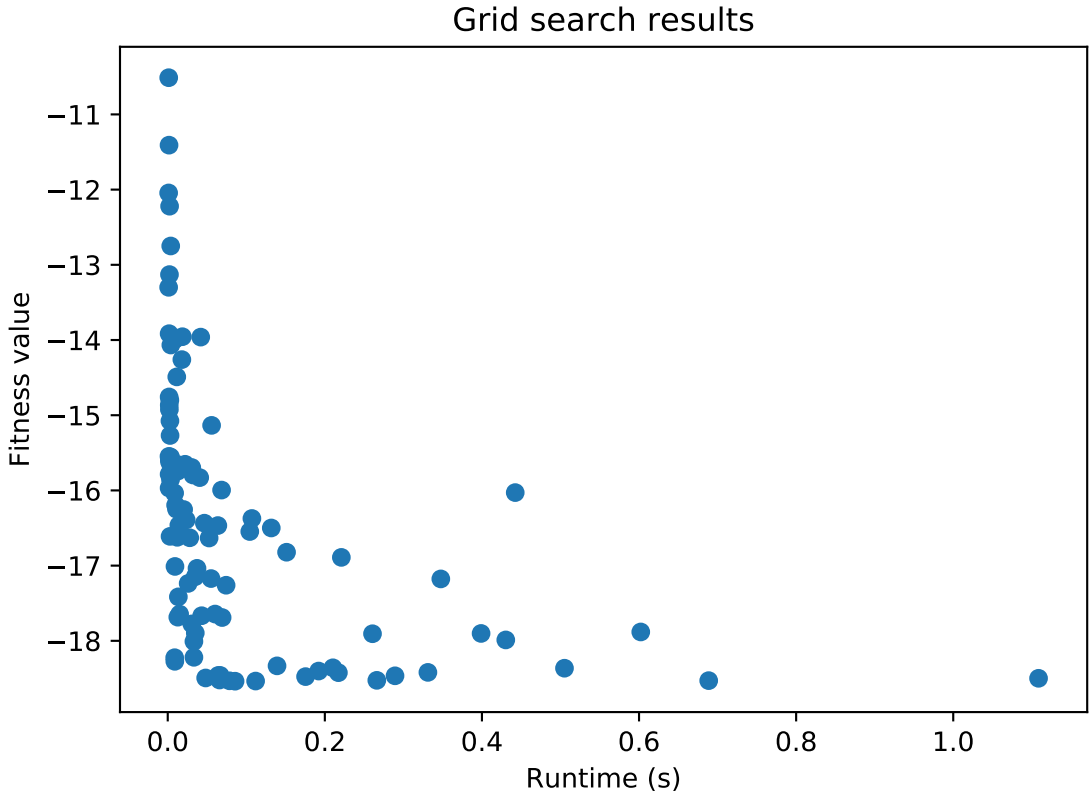
```
main('short', 3)
```

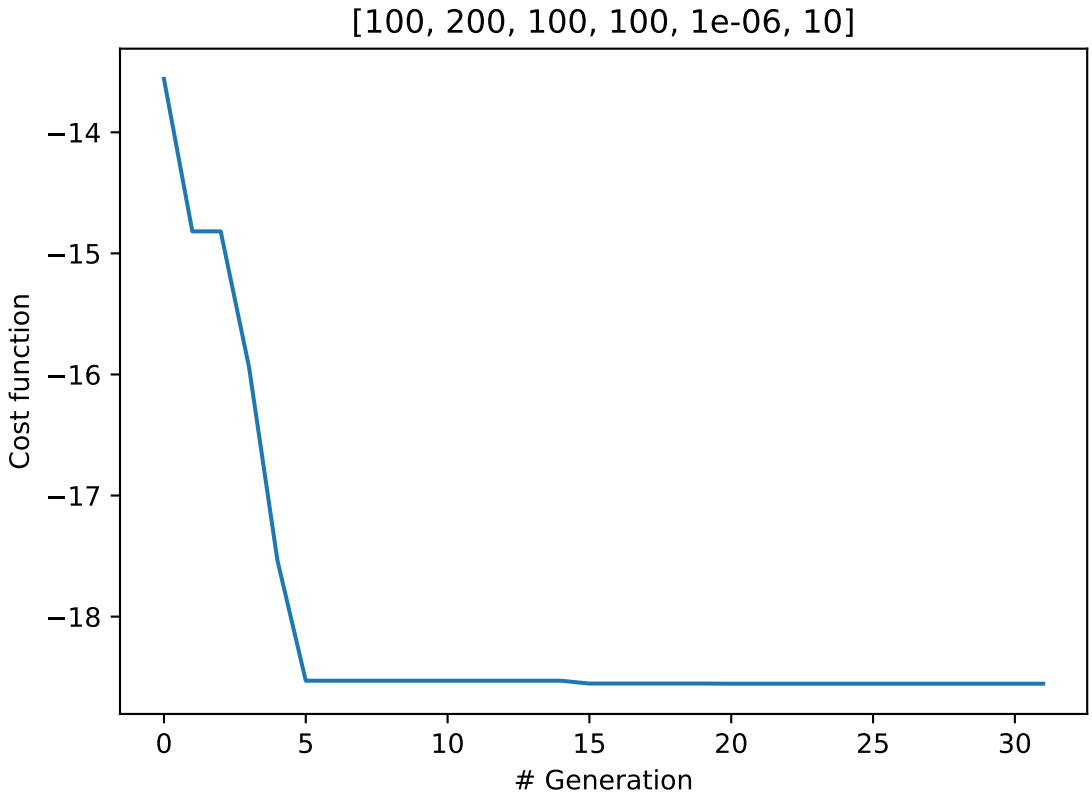
```
## Looking for best parameters:
##  · 96 combinations being tested.
##  · Results averaged using 3 repetitions.
## Best parameters found: [100, 200, 100, 100, 1e-06, 10]
##
## Building model...
## Generation: 0
##  Population size 100
##  Best plant [5.872639334788006, 8.977560537765859, -13.558995246887207]
```

```
##
## Generation: 1
##  Population size 200
##  Best plant [5.808339173313745, 8.565791399674527, -14.818090438842773]
##
## Generation: 2
##  Population size 200
##  Best plant [5.808339173313745, 8.565791399674527, -14.818090438842773]
##
## Generation: 3
##  Population size 200
##  Best plant [9.18502171423545, 8.42409758720295, -15.939918518066406]
##
## Generation: 4
##  Population size 200
##  Best plant [9.15651468274378, 8.707618377779891, -17.534791946411133]
##
## Generation: 5
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 6
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 7
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 8
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 9
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 10
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 11
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 12
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 13
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 14
```

```
##  Population size 200
##  Best plant [9.04608345359863, 8.634589005225937, -18.52950668334961]
##
## Generation: 15
##  Population size 200
##  Best plant [9.042930509921966, 8.674991348453016, -18.552709579467773]
##
## Generation: 16
##  Population size 200
##  Best plant [9.042930509921966, 8.674991348453016, -18.552709579467773]
##
## Generation: 17
##  Population size 200
##  Best plant [9.042930509921966, 8.674991348453016, -18.552709579467773]
##
## Generation: 18
##  Population size 200
##  Best plant [9.042930509921966, 8.674991348453016, -18.552709579467773]
##
## Generation: 19
##  Population size 200
##  Best plant [9.042930509921966, 8.674991348453016, -18.552709579467773]
##
## Generation: 20
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 21
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 22
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 23
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 24
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 25
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 26
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 27
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
```

```
##
## Generation: 28
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 29
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 30
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## Generation: 31
##  Population size 200
##  Best plant [9.041805156773096, 8.668185552670254, -18.554147720336914]
##
## 4.3762733936309814
```




[100, 200, 100, 100, 1e-06, 10]
−14
−15
−16
−17
−18
Cost function
0
5
10
15
20
25
30
Generation