Computação Evolucionária

2013/2014

Notes for the PL 2

Ernesto J. F. Costa

17 de Fevereiro de 2014

Class PL 2

Single-State Stochastic Algorithms

2.1 Algorithms

During the lectures we presented a class of algorithms collectively known as single-state stochastic algorithms. The name result from the fact they all work with a **single** individual which they try to iteratively improve. The process stops when we found a good solution or when we run out of time. If the new candidate solution is loosely related with the current one (or not related at all) we say that we are doing a **global search**, while if the new candidate is one of the neighbors of the current one we talk of **local search**. Finally, these candidates are produced **stochastically**. The algorithms discussed were the following:

- Random Search
- Hill-Climbing
- Hill-Climbing with Random Restart
- Simulated Annealing
- Tabu Search
- Iterated Local Search

Our goal now is to do some experiments with these algorithms (or some variations). In **Inforestudante** you will find a simple implementation of all of them. It must be clear that these implementations are in part problem dependent. Let's look to an example involving `Random Search` whose pseudo code is presented in algorithm 1.

It is not very hard to implement it in `Python` as the listing 2.1 shows.

Algoritmo 1: Random Search.

```
Input: NumIterations, Domain,ProblemSize,Cost
Output: Best
1 Best ← RandomSolution(ProblemSize,Domain);
2 foreach $iter_i$ ∈ NumIterations do
3     $candidate_i$ ← RandomSolution(ProblemSize, Domain);
4     if Cost($candidate_i$) < Cost(Best) then
5         Best ← $candidate_i$;
6 return Best;
```

```python
def random_search(domain,fitness,max_iter):
    """
    domain: [...,[xi_min, xi_max],...]
    """
    best = random_candidate_float(domain)
    cost_best = fitness(best)
    for i in range(max_iter):
        candidate = random_candidate_float(domain)
        cost_candi = fitness(candidate)
        if cost_candi < cost_best:
            best = candidate
            cost_best = cost_candi
    return best
```

Listagem 2.1: Random Search

The first thing to notice is the fact that we had to define a **representation** for the problem. Each candidate solution will be represented by a vector whose size is equal to the problem size. The values of each component of the candidate must be within an upper and a lower bound, both defined in the domain. Second, we assume that we are working in $\mathbb{R}^n$, and so the function that produces a candidate works with floats.

Let's turn our attention now to the problem of generating a candidate. The solution is trivial, for we just need to use `Python`'s list comprehensions, as we can see in listing 2.2.

```python
def random_candidate_float(domain):
    return [random.uniform(domain[i][0],domain[i][1]) for i in
        range(len(domain))]
```

Listagem 2.2: Random generation of a vector of floats

In order to test it we need a concrete problem. To illustrate, we choose to answer the question about finding the minimum of the DeJong's function **F1**, which is 0 at $x = 0$:

$$dejong_{f1}(x) = \sum_{i=1}^{n} x_i, \quad \text{with } x = (x_1, \ldots, x_i, \ldots, x_n)$$

This is not an hard problem, as we can infer from the figure of the function shown in 2.1, for the case of $n = 2$.

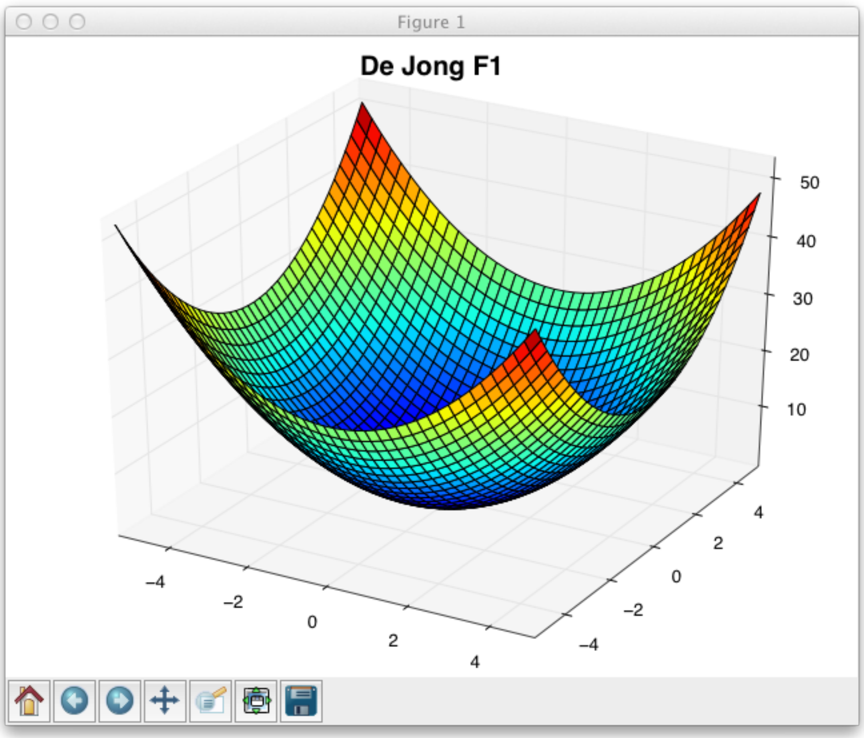


Figura 2.1: De Jong's F1

The fitness function is just the function's definition and, again, we do not have many trouble in implement it:

```python
def de_jong_f1(individual):
    """
    De Jong F1 or the sphere function
    domain: [-5.12, 5.12] for each dimension.
    min = 0 at x = (0,0,...,0)
    """
    return sum([ x_i ** 2 for x_i in individual])
```

Now we have everything we need to test the implementation with this simple example. Bellow we show the code complete.

```
import random
from math import exp, sin, cos, pi

def random_search(domain,fitness,max_iter):
    """
    domain: [...,[xi_min, xi_max],...]
    """
    best = random_candidate_float(domain)
    cost_best = fitness(best)
    for i in range(max_iter):
        candidate = random_candidate_float(domain)
        cost_candi = fitness(candidate)
        if cost_candi < cost_best:
            best = candidate
            cost_best = cost_candi
    return best

def random_candidate_float(sp):
    return [random.uniform(sp[i][0],sp[i][1]) for i in range(
       len(sp))]


def de_jong_f1(individual):
    """
    De Jong F1 or the sphere function
    domain: [-5.12, 5.12] for each dimension.
    min = 0 at x = (0,0,...,0)
    """
    return sum([ x_i ** 2 for x_i in individual])

if __name__ == '__main__':
    dimension = 3
    search_space = [[-5.12, 5.12] for i in range(dimension)]
    print(random_search(search_space,de_jong_f1,120))
```
Listagem 2.3: Looking for the minimum with random search

2.2 Benchmark Problems

In order to play with the different algorithms that we have introduced during the lectures we need some testbed problems. The ones presented here may be complemented by you.

Deadened Oscilations This is a function with many local maxima (and minima). The global maximum is at $x = 0$.

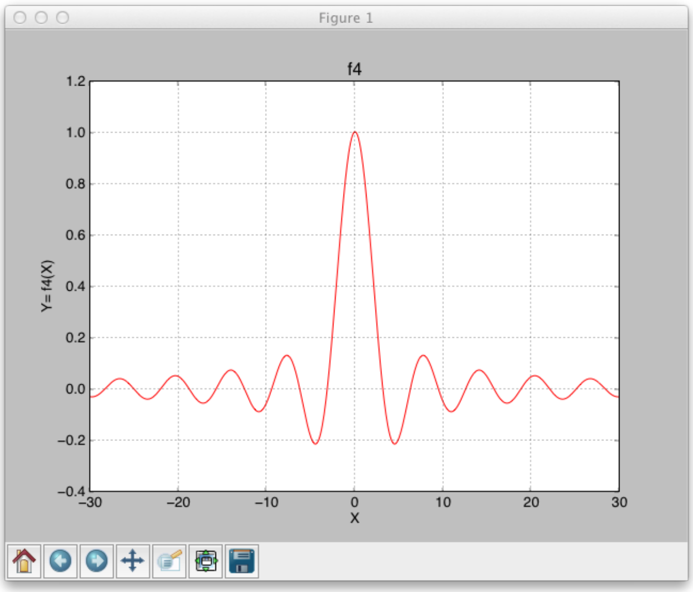


Figura 2.2: A função $f(x) = \frac{sin(x)}{x}$

Periodic Many global maxima (and minima). For example in the interval [-5,5] the maximum is at $x = -1.05$. To be notice also the existence of several saddle points.

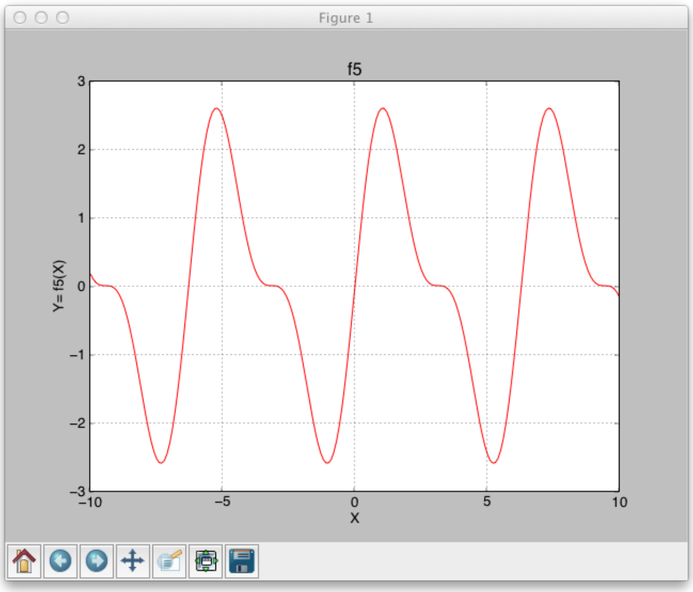


Figura 2.3: A função $f(x) = 2 \times sin(x) + sin(2 \times x)$

Michalewicz Similar to De Jong's F1 with a maximum equal to at 2.85 at $x = 1.85$ in the interval [-1,2].

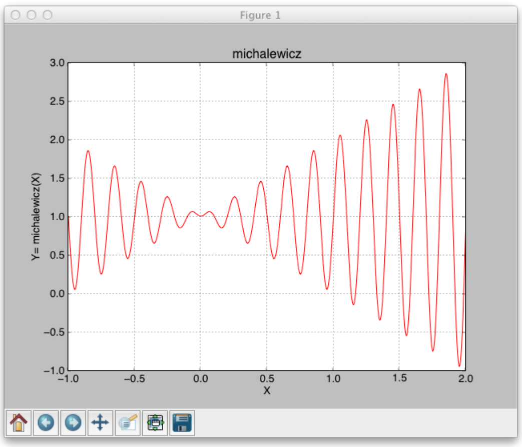


Figura 2.4: A função $f(x) = x \times sin(10 \times \pi \times x) + 1$

Trap This the example of a deceptive function. The fitness is equal to the number of zeros in a binary string of size n. Nevertheless if there are no zeros the value of the fitness is $(n+1)$. The definition is given by 2.1.

$$f(\langle x_1,\dots,x_n\rangle) = (n - \sum_{i=1}^{n} x_i) + (n+1)\prod_{i=1}^{n} x_i \qquad x_i \in \{0,1\} \tag{2.1}$$

Rosenbrock Also known as De Jong's F2. It is continuous, non convex and unimodal. The 2D version can be seen in figure 2.5.

$$Min(F_2) = F_2(1,\dots,1) = 0$$

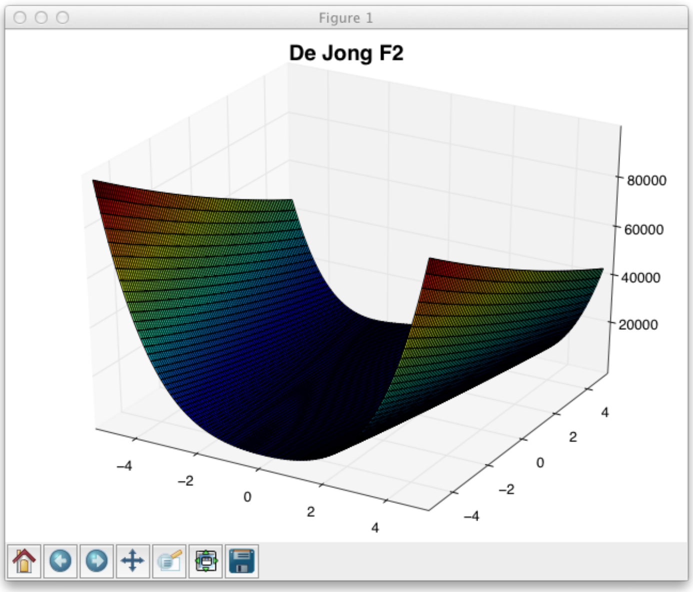


Figura 2.5: **De Jong F2**

It is defined by the equation 2.2.

$$f(\langle x_1,\dots,x_n\rangle) = \sum_{i=1}^{n-1}(1-x_i)^2 + 100\times(x_{i+1}-x_i^2)^2 \qquad x_i \in [-2.048, 2.048] \tag{2.2}$$

Quartic Also known as D Jong's F4. De Jong F4. Continuous, convex, unimodal (see figure 2.6).

$$Min(F_4) = F_4(0, \ldots, 0) = 0$$

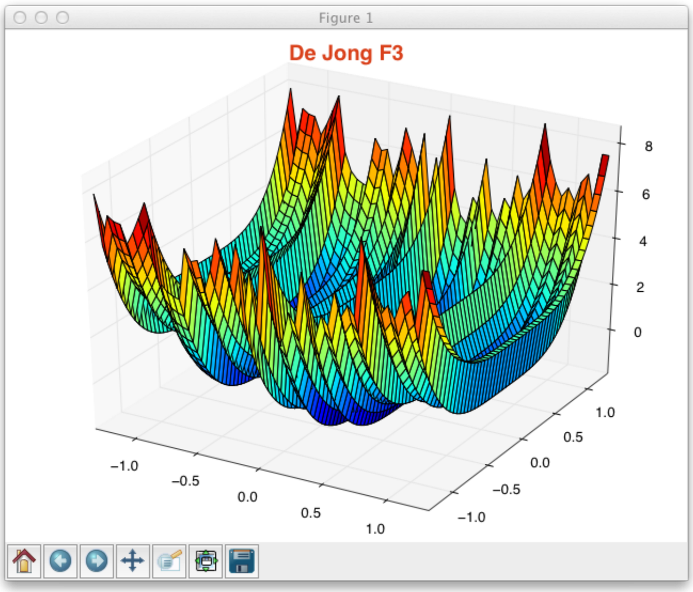


Figura 2.6: **De Jong F4**

It is defined by the equation 2.3.

$$f(\langle x_1, \ldots, x_n \rangle) = (\sum_{i=1}^{n} i \times x_i^4) + \mathcal{N}(0,1) \qquad x_i \in [-1.28, 1.28] \tag{2.3}$$

with $\mathcal{N}(0,1)$ a gaussian noise of mean 0 and standard deviation 1.

Rastrigin Highly multimodal with many local minima (see figure 2.7 for a 2D version).

$$Min(Rastrigin) = Rastrigin(0, \ldots, 0) = 0$$

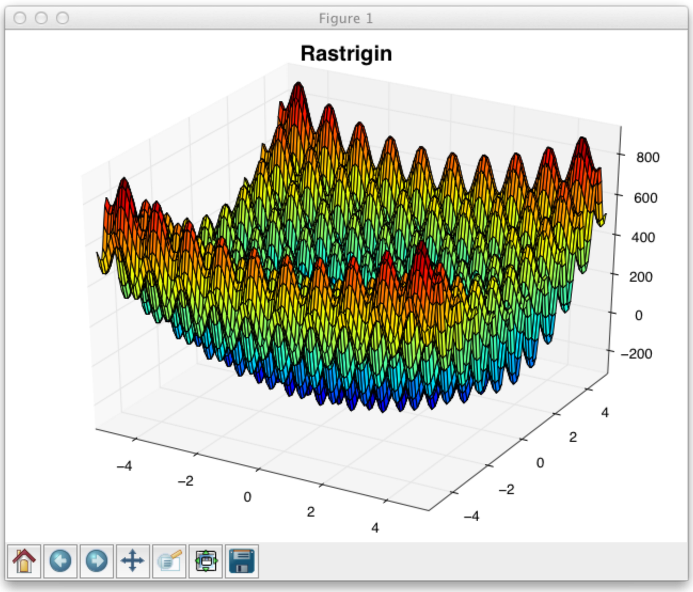


Figura 2.7: **Rastrigin**

It is defined by the equation 2.4.

$$f(\langle x_1, \dots, x_n \rangle) = A \times n + (\sum_{i=1}^{n} x_i^2 - A \times cos(2\pi x_i) \qquad x_i \in [-5.12, 5.12] \quad (2.4)$$

with A typically equal to 10.

Schwefel Multimodal with many local minima (see figure 2.8 for a 2D version).

$$Min(Schwefel) = Schwefel(420.9687, \dots, 420.9687) = n \times 418.9829$$

It is defined by the equation 2.5.

$$f(\langle x_1, \dots, x_n \rangle) = \sum_{i=1}^{n} -x_i \times sin(\sqrt{|x_i|}) \qquad x_i \in [-500, 500] \qquad (2.5)$$

Schubert

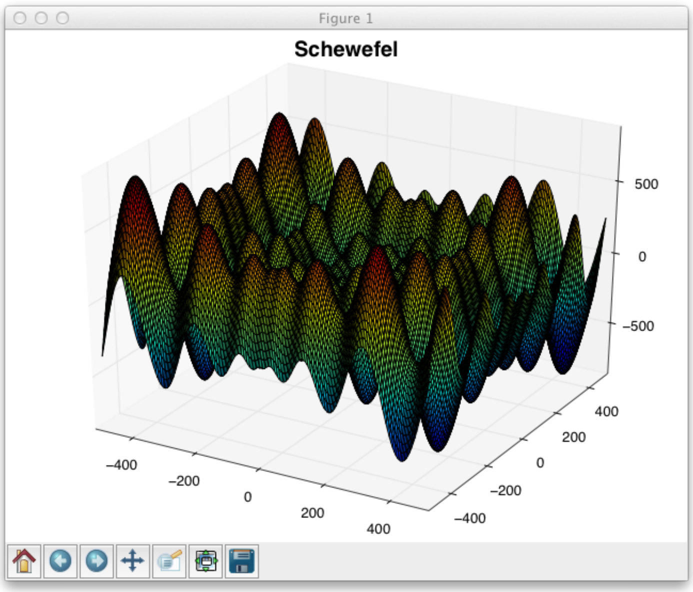


Figura 2.8: **Schwefel**

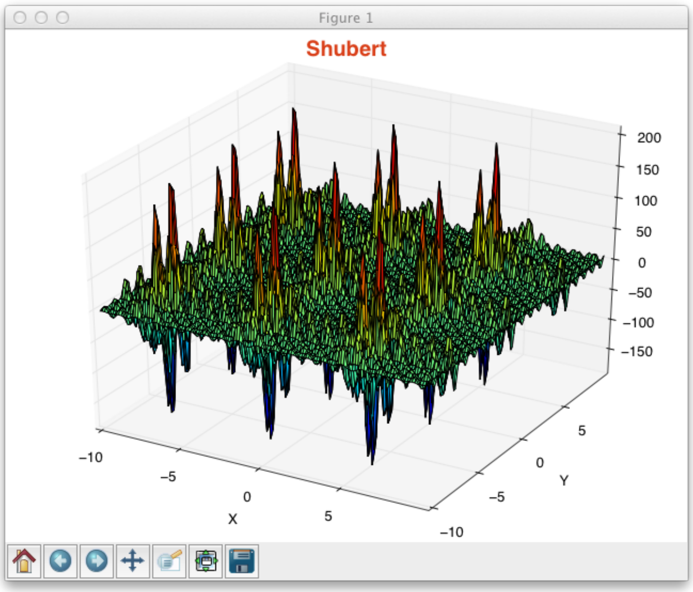


Figura 2.9: **Shubert**

2.3 Let's do it

2.3.1 Warmup

Take the code given and do some experiments with the different algorithms. You may choose just one function (e.g., deadened oscillations) and try to compare the performance of the different alternatives. But remember that it is rewarding to use different **types** of examples/functions. An algorithm may be good in some cases and very bad in others: there is No Free Lunch! Also pay attention to the values of the different parameters used in the algorithms.

It is very informative to visualize the evolution of the candidate solution over time. Store these values and in the end use the module `matplotlib` to plot these values. In **Inforestudante** you will find some examples showing how you can do that easily.

2.3.2 Design

Take the hill-climbing algorithm and change it, so that instead of choosing one neighbor (eventually a good one) choose the best of the neihgbors. Is this easily achievable for all kind of problems (domains)? Test your implementation.

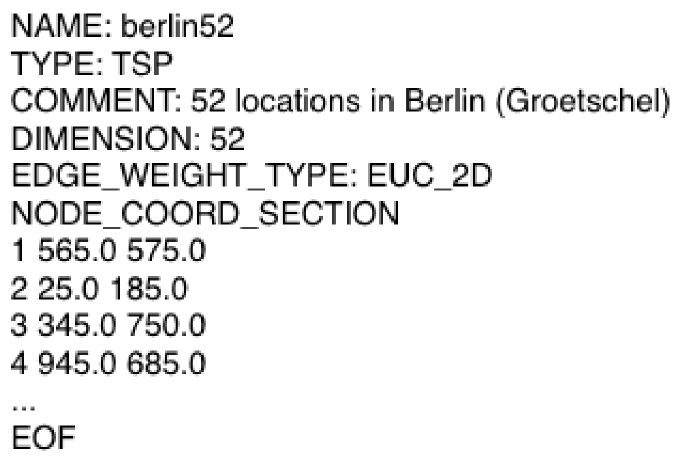
2.3.3 Like a weasel

There is a dialog in the play *Hamlet*, by W. Shakespeare, where Hamlet says to Polonius: *Methinks it is like a weasel.* Many years later the biologist Richard Dawkings used that sentence in his book *The blind watchmaker* to show the power of cumulative natural selection in the evolution of species. Our goal in this experiment is to study the possibility of our single-state stochastic algorithms to discover that sentence starting from a random generated sentence of the same length.

2.3.4 Explorations

We have talked of the Traveling Salesman Problem (TSP): a person that has to visit several cities starting from his home town, one and just once, and then return to his/her starting city,. The tour must have a minimum (distance) cost. Adapt the provided code so you can solve it. Compare the performance of the several algorithms when applied to this problem.

You are going to need a file with the description of a concrete example. There are some examples in **Inforestudante**. Notice that these files have a specific **format** as it is illustrated in the figure 2.10.



```
NAME: berlin52
TYPE: TSP
COMMENT: 52 locations in Berlin (Groetschel)
DIMENSION: 52
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
4 945.0 685.0
...
EOF
```

Figura 2.10: Example of the format of a tsp file. You may have more than one COMMENT line.

For the purpose of the exercise, the only relevant part are the 2D coordinates of each city.