Evolutionary Computation

Notes for the PL 2

Ernesto Costa
Nuno Lourenço
João Macedo

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 results 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 **UCStudent** you will find simple implementations for some of them. It must be clear that these implementations are in part problem dependent.

2.2 Random Search

To warm up let's look to an example involving `Random Search` whose pseudo code is presented in algorithm 1.

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;
```

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

```
1  def random_search(domain,fitness,max_iter):
2      """
3      domain = [...,[xmin_1, xmax_i],...]
4      fitness = how to quantify the quality of a candidate
           solution
5      """
6      best = random_indiv(domain)
7      cost_best = fitness(best)
8      for i in range(max_iter):
9          candidate = random_indiv(domain)
10         cost_candi = fitness(candidate)
11         if cost_candi < cost_best: # minimization
12             best = candidate
13             cost_best = cost_candi
14     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 dimension of the problem. 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.

```
1 def random_indiv(domain):
2     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**, also known as the **Sphere Function**, which is 0 at $x = 0$[1]:

$$dejong_{f1}(x) = \sum_{i=1}^{n} x_i^2, \quad \text{with } x = (x_1, \ldots, x_i, \ldots, x_n)$$

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

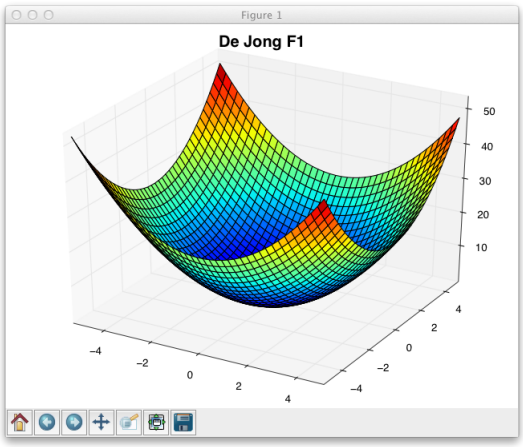

Figura 2.1: De Jong's F1

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

```
def dejong_f1(individual):
    """
    De Jong F1 or the sphere function
```

[1] x is in fact a n-dimensional vector, so to be formally correct we should have said $\vec{x} = (x_1, \ldots, x_n)$.

```
    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 complete code.

```
1  import random
2
3  def random_search(domain,fitness,max_iter):
4      """
5      domain = [...,[xmin_1, xmax_i],...]
6      fitness = how to quantify the quality of a candidate
           solution
7      """
8      best = random_indiv(domain)
9      cost_best = fitness(best)
10     for i in range(max_iter):
11         candidate = random_indiv(domain)
12         cost_candi = fitness(candidate)
13         if cost_candi < cost_best: # minimization
14             best = candidate
15             cost_best = cost_candi
16     return best
17
18  def random_indiv(domain):
19     return [random.uniform(domain[i][0],domain[i][1]) for i in
           range(len(domain))]
20
21  def dejong_f1(individual):
22     """
23     De Jong F1 or the sphere function
24     domain: [-5.12, 5.12] for each dimension.
25     min = 0 at x = (0,0,...,0)
26     """
27     return sum([ x_i ** 2 for x_i in individual])
28
29 if __name__ == '__main__':
30     dimension = 3
31     search_space = [[-5.12, 5.12] for i in range(dimension)]
```

```
32 print(random_search(search_space,dejong_f1,120))
```

Listagem 2.3: Looking for the minimum with random search

2.3 João Brandão Numbers' (JBN): a benchmark problem

We discussed this problem in class. Let's remind it. Suppose that we have the set of all integers between 1 and a certain n,i.e., $D = \{1, 2, \ldots, n\}$. The idea is to find one sub-set $JB \subset D$ of maximum size, and such that there is not a number is the subset that is the average of other two members in that same sub-set. Remember, it's a set, so repetitions are not allowed. For example, if $n = 6$, then the set $\{1, 3, 4, 6\}$ is a possible solution. In class, we also saw how to implement a solution based on the standard Hill-climbing algorithm (see 2.4.).

```
"""
Numbers of João Brandão.

Algoritmo: Hill-climbing
Perturbation: best neighbor
Representation: binary

"""

import random

# Basic Hill Climbing
def jb_hc(problem_size, max_iter,fitness):
    candidate = random_indiv(problem_size)
    cost_candi = fitness(candidate)
    for i in range(max_iter):
        next_neighbor = best_neighbor(candidate,fitness)
        cost_next_neighbor = fitness(next_neighbor)
        if cost_next_neighbor >= cost_candi:
            candidate = next_neighbor
            cost_candi = cost_next_neighbor
    return candidate

# Random Individual
```

```
def random_indiv(size):
    return [random.randint(0,1) for i in range(size)]

# Best neighbor
def best_neighbor(individual, fitness):
    best = individual[:]
    best[0] = (best[0] + 1) % 2
    for pos in range(1,len(individual)):
        new_individual = individual[:]
        new_individual[pos]= (individual[pos] + 1) % 2
        if fitness(new_individual) > fitness(best):
            best = new_individual
    return best

# Fitness for JB
def evaluate(indiv):
    alfa = 1
    beta = 1.5
    return alfa * sum(indiv) - beta * viola(indiv)

def viola(indiv):
  # count constraint violations
  comp=len(indiv)
  v=0
  for i in range(1,comp):
    limite= min(i,comp - i - 1)
    vi=0
    for j in range(1,limite+1):
      if (indiv[i]==1) and (indiv[i-j]==1) and (indiv[i+j] ==
         1):
        vi+=1
    v+=vi
  return v

# Auxiliar
def fenotipo(indiv):
    fen = [i+1 for i in range(len(indiv)) if indiv[i] == 1]
    return fen


if __name__ == '__main__':
```

```
# For test purposes: beware of the time it may takes...
best = jb_hc(10,100,evaluate)
res = fenotipo(best)
quali = viola(best)
print('INDIV: %s\nQUALIDADE: %s\nTAMANHO:%s' % (res, quali
    ,len(res)))
```

Listagem 2.4: João Brandão's Number by Hill-climbing

First Experiments Let's do some experiments with the standard hill-climbing problem and the following variants:

- Hill-Climbing choosing the **first neighbor** that is better than the current one;
- Hill-Climbing with Random Restart.

Run the three versions with different values of n and compare the results.

2.4 Me thinks ...

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 Dawkins used that sentence in his book *The blind watchmaker* to show the power of cumulative natural selection in the evolution of species. In class we showed a possible Python implementation for the problem based on a single-state stochastic algorithm. Our goal in this experiment is to study the implications of the tweaking mechanism. For example, what is the importance of the size of the neighborhood? What if you have a **global** approach, i.e., you replace a component of the candidate solution (a letter) by another one chosen randomly from the alphabet?

2.5 Challenge: Other Algorithms

Some of the algorithms studied in class were left behind until now, namely simulated annealing, tabu search and iterated local search. It is time to catch-up and try to implement them. We will analise their behavior using the JB problem. Try with different values of n. Below you will find the pseudo code description of each of them.

2.5.1 Simulated Annealing

In Algorithm 2 we present the pseudo code for the Simulated Annealing Algorithm.

Algoritmo 2: Simulated Annealing.

Input: ProblemSize, $iterations_{max}$, $temp_{max}$
Output: S_{best}

```
1  S_current ← CreateInitialSolution(ProblemSize);
2  S_best ← S_current;
3  for i = 1 to iterations_max do
4      S_i ← CreateNeighborSolution(S_current);
5      temp_curr ← CalculateTemperature(i, temp_max);
6      if Cost(S_i) ≤ Cost(S_current) then
7          S_current ← S_i;
8          if Cost(S_i) ≤ Cost(S_best) then
9              S_best ← S_i;
10     else if Exp( (Cost(S_current) − Cost(S_i)) / temp_curr ) > Rand() then
11         S_current ← S_i;
12 return S_best;
```

2.5.2 Tabu Search

In Algorithm 3 we present the pseudo code for the Tabu Search Algorithm.

Algoritmo 3: Tabu Search.

Input: $TabuList_{size}$
Output: S_{best}

```
1  S_best ← ConstructInitialSolution();
2  TabuList ← ∅;
3  while ¬ StopCondition() do
4      CandidateList ← ∅;
5      for S_candidate ∈ Sbest_neighborhood do
6          if ¬ ContainsAnyFeatures(S_candidate, TabuList) then
7              CandidateList ← S_candidate;
8      S_candidate ← LocateBestCandidate(CandidateList);
9      if Cost(S_candidate) ≤ Cost(S_best) then
10         S_best ← S_candidate;
11         TabuList ← FeatureDifferences(S_candidate, S_best);
12         while TabuList > TabuList_size do
13             DeleteFeature(TabuList);
14 return S_best;
```

2.5.3 Iterated Local Search

In Algorithm 4 we present the pseudo code for the Iterated Local Search Algorithm.

Algoritmo 4: Iterated Local Search.

Input:
Output: S_{best}

```
1  S_best ← ConstructInitialSolution();
2  S_best ← LocalSearch();
3  SearchHistory ← S_best;
4  while ¬ StopCondition() do
5      S_candidate ← Perturbation(S_best, SearchHistory);
6      S_candidate ← LocalSearch(S_candidate);
7      SearchHistory ← S_candidate;
8      if AcceptanceCriterion(S_best, S_candidate, SearchHistory) then
9          S_best ← S_candidate;
10 return S_best;
```