Exercício 05

Renan Salles de Freitas
CPE 723 - Otimização Natural

7 de maio de 2018

Exercício 1. Segue o código MatLab correspondente ao EP para resolver a função de Ackley para $n = 30$, $\mu = 200$:

```
clear all
clc

J = [];

for run=1:100

    global tau tau_prime epsilon population_size q

    number_of_iteration = 1000;
    number_of_states = 30;
    population_size = 200;
    tau = 1/sqrt(2 * sqrt(number_of_states));
    tau_prime = 1/sqrt(2 * number_of_states);
    epsilon = 1e-2;
    q = 10;
    jmin = inf;

    s = randn(number_of_states, population_size);
    pop = (30+30)*rand(number_of_states, population_size)-30;

    n = 1;
    while n < number_of_iteration + 1

        % Mutacao
        [new_pop, new_s] = mutation(pop, s);

        children = [pop , new_pop];
        children_s = [s , new_s];

        j = f(children);
        jm = min(j);
```

```
        if jm < jmin
            jmin = jm;
        end
        [pop, s] = selection(j, children, children_s);
        n = n + 1;
    end
    J = [J, jmin];
end
```

Para a mutação, optou-se pela não correlacionada de n steps:

```
function [y, s] = mutation(x, s)
global tau tau_prime epsilon population_size

s = s.*exp(tau_prime * randn(1, population_size) + tau * randn(size(x
    )));
s(s<epsilon) = epsilon;

y = x + s.*randn(size(x));
```

Todos sofrem mutação.

A seleção ocorre com o critério *round-robin tournament* com $q = 10$, para o conjunto pais e filhos:

```
function [pop, s] = selection(j, children, children_s)
global population_size q

children_size = size(children, 2);
games = zeros(q, children_size);

for game=1:q
    p = randperm(children_size);
    for match=1:children_size/2
        player_1 = p(match * 2 - 1);
        player_2 = p(match * 2);
        if j(player_1) < j(player_2)
            games(game, player_1) = 1;
        elseif j(player_1) > j(player_2)
            games(game, player_2) = 1;
        end
    end
end

games = sum(games, 1);

[~, jsort_index]=sort(games, 'descend');
jsort_index = jsort_index(1:population_size);

pop = children(:,jsort_index);
```

```
26 s = children_s(:,jsort_index);
```

Avaliando a implementação 100 vezes e calculando-se a média e desvio padrão, obtemos:

$$J_{\text{medio}} = 16.4641$$
$$J_{\text{std}} = 2.4430$$
$$J_{\text{min}} = 2.1270$$

Mostrando que a recombinação é importante para este problema, já que o desempenho foi bem pior.

Exercício 2. Na implementação dos EA clássicos (baunilha) o número de avaliações da função custo é fixo em cada geração. Por exemplo, no exercício 1 acima, o número de avaliações da função custo é 2 × tamanho da população por geração. Dessa forma, nos algoritmos clássicos, avaliar a velocidade do EA por número de gerações é semelhante a avaliá-lo pelo número de vezes que a função custo é computada.

Exercício 3. Os algoritmos genéticos buscam um equilíbrio entre exploração global e local, isto é, explorar a área de busca tanto quanto possível, e concentrar a exploração em torno de um ponto (mínimo global).

Em algoritmos genéticos, a mutação é o passo que tenta evitar a convergência e explorar mais a área de busca. Nos algoritmos, faz sentido explorar mais a área de busca no início, nas gerações iniciais (assegurando a diversidade da população) e, nas últimas gerações, o algoritmo tenta reduzir a área de busca, explorando próximo ao mínimo global. Nessa estratégia, o parâmetro de mutaçãoo cai com o número de gerações. Há, porém, um problema comum nessa estratégia. Quando a população converge para um mínimo local, o algoritmo deveria aumentar a diversidade da população, explorando outras áreas. Esses dois comportamentos são justificativas de porque os parâmetros da mutaçãoo devem ser aumentados e reduzidos conforme as gerações.

Exercício 4. Foi implementado um ES para resolver o problema de *clustering* em $\mathbb{R}^2$. O código MatLab segue abaixo:

```
1  clear all
2  clc
3
4
5  global tau tau_prime epsilon population_size
6
7  map_size = 10;
8  number_of_clusters = 2;
9  number_of_points_per_cluster = 20;
10 cluester_threshold = 2;
11 data_threshold = 0.1;
12
13 clusters = define_clusters(map_size, number_of_clusters,
       cluester_threshold);
```

```
data = generate_data(clusters, number_of_points_per_cluster,
   data_threshold);

number_of_iteration = 1000;
population_size = 30;
number_of_parents = 100;
tau = 0.1/sqrt(2 * sqrt(number_of_clusters * 2));
tau_prime = 0.1/sqrt(4 * number_of_clusters);
epsilon = 1e-2;
jmin = inf;

s = randn(2 * population_size, number_of_clusters);
pop = rand(2 * population_size, number_of_clusters) * map_size;
best_cluster = [];

n = 1;
while n < number_of_iteration+1

    % Recombinacao
    children = zeros(2 * number_of_parents, number_of_clusters);
    children_s = zeros(2 * number_of_parents, number_of_clusters);
    i = 1;
    while i < number_of_parents
        index_1 = randi(population_size);
        index_2 = randi(population_size);
        if index_1 == index_2
            continue
        end
        children(i:i+1,:) = ...
            crossover_discrete(pop(index_1 * 2 - 1:index_1 * 2,:),
               ...
            pop(index_2 * 2 - 1:index_2 * 2,:));
        children_s(i:i+1,:) = ...
            crossover_global_intermediate( ...
            s(index_1 * 2 - 1:index_1 * 2,:), ...
            s(index_2 * 2 - 1:index_2 * 2,:));
        i = i + 1;
    end

    % Mutacao
    [children, children_s] = mutation(children, children_s);

    j = f(children, data);
    [jm, index] = min(j);
    if jm < jmin
        jmin = jm;
        best_cluster = children(index * 2 - 1:index * 2,:);
```

```
59      end
60      [pop, s] = comma_selection(j, children, children_s);
61      n = n + 1;
62  end
```

Os parâmetros escolhidos podem ser vistos no código. Após 1000 gerações, obtivemos $J_{\min} = 0.3889$. A figura abaixo mostra a situação final.