Exercício 04

Renan Salles de Freitas
CPE 723 - Otimização Natural

27 de abril de 2018

Exercício 1. O objetivo deste exercício é escrever um programa de algoritmo genético simples. O código de MatLab está abaixo:

```
clear all
clc

number_of_iteration = 100;
n = 1;
number_of_bits = 16;
population_size = 30;
xbimax = 2 ^ number_of_bits - 1;
pc = 0.7;
pm = 0.001;
jmax = 0;
xmax = 0;
xdemin = -2;
xdemax = 2;

pop = randi(2, population_size, number_of_bits) - 1;


while n <number_of_iteration
    xr = mapbi2func(pop, xdemin, xdemax, xbimax);
    j = -f(xr);

    [jm, index] = max(j);
    if jm > jmax
        jmax = jm;
        xmax = xr(index);
    end

    mating_pool = sus_selection(pop, j);
    children = zeros(population_size, number_of_bits);

    % Recombinacao
```

```
33      i = 1;
34      while i <= population_size
35          if rand < pc && i < population_size - 1
36              [children(i,:), children(i+1,:)] = ...
37                  crossover(mating_pool(i,:), mating_pool(i+1,:));
38              i = i + 2;
39          else
40              children(i,:) = mating_pool(i,:);
41              i = i + 1;
42          end
43      end
44
45      % Mutacao
46      for i = 1:population_size
47          for j = 1:length(children(i))
48              if rand < pm
49                  children(i,:) = mutation(children(i,:), j);
50              end
51          end
52      end
53
54      pop = children;
55      n = n + 1;
56
57  end
```

Como o código apresenta diversas funções e há várias maneiras de desenvolver o código, é nencessário explicar comoo cada etapa do GA foi implementada. Como parâmetros iniciais do algoritmoo, foi escolhida uma populaçãoo inicial de 10 indivíduos de 16 bits, com bit de maior valor significativo na esquerda. O fenótipo é a conversão do gennótipo em números reais entre -2 e 2 e avalia-se a função que se quer minimizar: $y = x^2 - 0.3\cos(10\pi x)$. O código MatLab para o fenótipo está abaixo:

```
1  function r = mapbi2func(x, xdemin, xdemax, xbimax)
2  xde = bi2dec(x);
3  r = (xdemax - xdemin) * xde / xbimax + xdemin;
```

```
1  function y = bi2dec(x)
2  x = flip(x);
3  y = bi2de(x);
```

```
1  function y = f(x)
2  y = x.^2 - 0.3 * cos(10 * pi * x);
```

Como é natural nos algoritmos genéticos, busca-se a minimização, portanto o valor da função é negativado (linha 21 do código do exercício 1). Na próxima etapa, guarda-se a melhor aptidão e o valor de x real correspondente.

Na seleção de pais, optou-se por utilizar o algoritmo *Amostragem Estocástica Universal* (SUS). O ranking é feito com probabilidade, de acordo com a aptidão. A implementaçãoo está abaixo:

```
function mating_pool = sus_selection(x, j)
[xsort, psort] = ranking(x, j);
number_of_bits = size(xsort, 2);
population_size = size(xsort, 1);
current_member = 1;
i=1;

mating_pool = zeros(population_size, number_of_bits);
psort = cumsum(psort);

r = rand / population_size;
while current_member <= population_size
    while r <= psort(i)
        mating_pool(current_member,:) = xsort(i,:);
        r = r + 1/population_size;
        current_member = current_member + 1;
    end
    i = i + 1;
end
```

```
function [xsort, psort] = ranking(x, j)
j = j + max(abs(j));
sumj = sum(j);
if sumj == 0
    p = ones(length(j), 1) / length(j);
else
    p = j / sum(j);
end
[psort, psort_index]=sort(p, 'descend');
xsort = x(psort_index,:);
```

A partir da geração de pais, a recombinação é feita se um um valor randômico for menor que $p_c = 0.7$. Se isso acontecer, dois pais geram dois filhos por recombinação do tipo 1 ponto (onde há troca da cauda dos bits). Aqui, optou-se por representação binária Gray. Códigoo abaixo:

```
function [nx1, nx2] = crossover(x1, x2)
% One point crossover
x1g = bi2gray(x1);
x2g = bi2gray(x2);
index = randi(length(x1));
temp = x1g(index:end);
x1g(index:end) = x2g(index:end);
x2g(index:end) = temp;
nx1 = gray2bi(x1g);
```

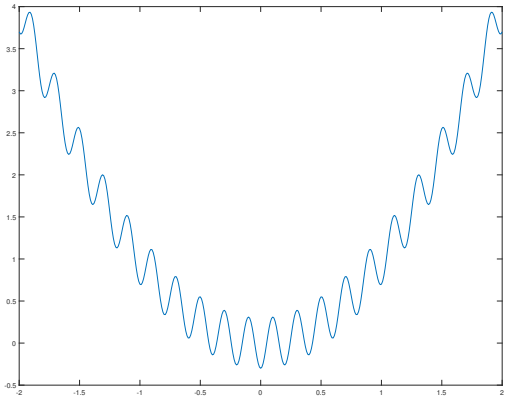
```
nx2 = gray2bi(x2g);
```

Caso o valor radômico não seja maior que p_c, o filho é uma cópia do pai.

Por fim, a mutação ocorre caso um valor randômico seja menor que $p_m = 0.001$. Aqui também optou-se pela codificação Gray, já que alterações de um bit representa números próximos em decimais. Códigoo abaixo:

```
function y = mutation(x, index)
xg = bi2gray(x);
xg(index) = xor(xg(index), xg(index));
y = gray2bi(xg);
```

A função pode ser vista abaixo:



Com uma população inicial de 10 indivíduos e realizando 100 iterações, obtemos população final: $x = 0.0023$ e $f(x) = -0.2992$.

Exercício 2. Exercício semelhante ao anterior, porém mudando parâmetros de população inicial para 100, número de bits 25, função custo a ser maximizada (não minimizada) e, por se tratar de uma função que usa a variável binária, não é necessário utilizar a codificação Gray. Os códigos MatLab e gráficos são apresentados abaixo:

```
clear all
clc

number_of_iteration = 100;
number_of_bits = 25;
population_size = 100;
xbimax = 2 ^ number_of_bits - 1;
pc = 0.7;
pm = 1/number_of_bits;
jmax = [];
jmin  = [];
jmean = [];

```

```
pop = randi(2, population_size, number_of_bits) - 1;

n = 1;
while n < number_of_iteration+1
    j = f(pop);

    jmax = [jmax ; max(j)];
    jmin = [jmin ; min(j)];
    jmean = [jmean ; mean(j)];

    mating_pool = sus_selection(pop, j);
    children = zeros(population_size, number_of_bits);

    % Recombinacao
    i = 1;
    while i <= population_size
        if rand < pc && i < population_size - 1
            [children(i,:), children(i+1,:)] = ...
                crossover(mating_pool(i,:), mating_pool(i+1,:));
            i = i + 2;
        else
            children(i,:) = mating_pool(i,:);
            i = i + 1;
        end
    end

    % Mutacao
    for i = 1:population_size
        for j = 1:length(children(i))
            if rand < pm
                children(i,:) = mutation(children(i,:), j);
            end
        end
    end

    pop = children;
    n = n + 1;

end
```

```
function y = f(x)
y = sum(x,2);
```

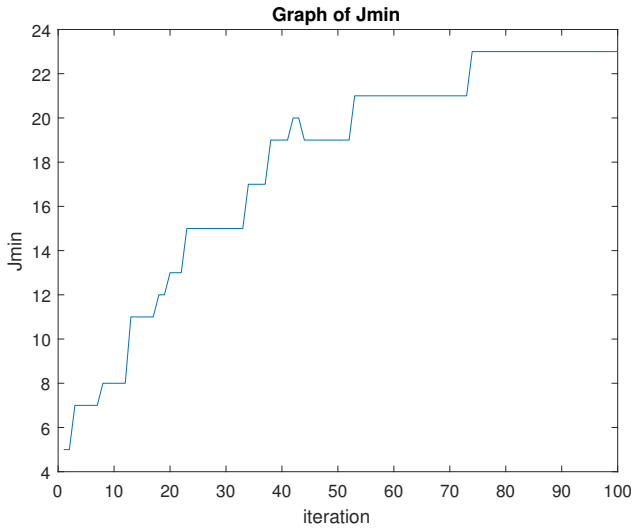
```
function mating_pool = sus_selection(x, j)
[xsort, psort] = ranking(x, j);
number_of_bits = size(xsort, 2);
population_size = size(xsort, 1);
current_member = 1;
```

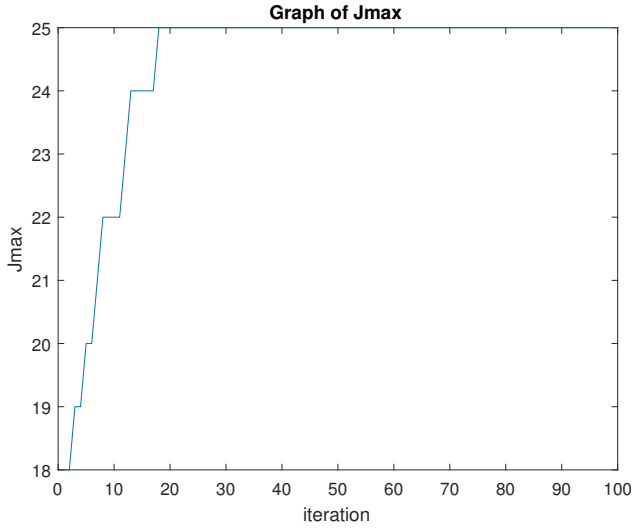
```
i=1;

mating_pool = zeros(population_size, number_of_bits);
psort = cumsum(psort);

r = rand / population_size;
while current_member <= population_size
    while r <= psort(i)
        mating_pool(current_member,:) = xsort(i,:);
        r = r + 1/population_size;
        current_member = current_member + 1;
    end
    i = i + 1;
end
```

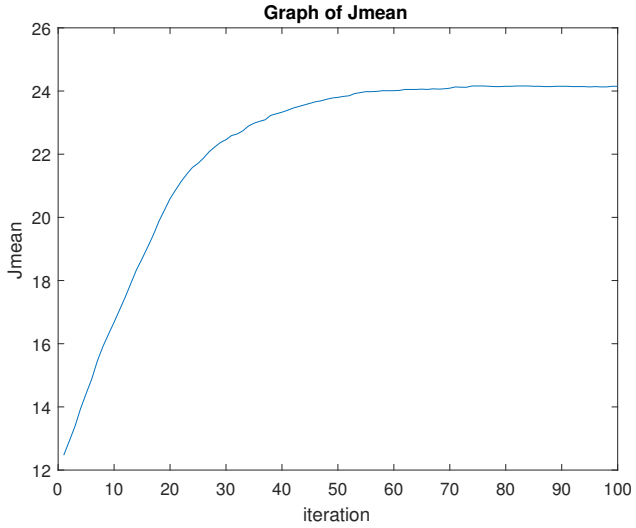
```
function [xsort, psort] = ranking(x, j)
sumj = sum(j);
if sumj == 0
    p = ones(length(j), 1) / length(j);
else
    p = j / sum(j);
end
[psort, psort_index]=sort(p, 'descend');
xsort = x(psort_index,:);
```

```
function [x1, x2] = crossover(x1, x2)
% One point crossover
index = randi(length(x1));
temp = x1(index:end);
x1(index:end) = x2(index:end);
x2(index:end) = temp;
```

```
function y = mutation(x, index)
x(index) = xor(x(index), x(index));
y = x;
```


Graph of Jmin
Jmin
iteration
4
6
8
10
12
14
16
18
20
22
24
0
10
20
30
40
50
60
70
80
90
100


Graph of Jmax
Jmax
iteration
18
19
20
21
22
23
24
25
0
10
20
30
40
50
60
70
80
90
100


Graph of Jmean
Jmean
iteration
12
14
16
18
20
22
24
26
0
10
20
30
40
50
60
70
80
90
100

A segunda parte do problema é rodar o algoritmo 10 vezes e verificar a média e desvio padrão de iterações para o programa achar o ótimo global. Foi encontrada média 19.2 e desvio padrão 2.7406.

Exercício 3. Segue o código MatLab correspondente ao algoritmo genético para resolver a função de Ackley para $n = 30$:

```
clear all
clc


global tau tau_prime epsilon population_size

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

    % Recombinacao
    children = zeros(number_of_states, number_of_parents);
    children_s = zeros(number_of_states, number_of_parents);
    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) = crossover_discrete(pop(:,index_1), pop(:,
           index_1));
        children_s(:,i) = ...
            crossover_global_intermediate(s(:,index_1), s(:,index_1))
               ;
        i = i + 1;
    end

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

```
40
41      j = f(children);
42      jm = min(j);
43      if jm < jmin
44          jmin = jm;
45      end
46      [pop, s] = comma_selection(j, children, children_s);
47      n = n + 1;
48  end
```

Para a recombinação, escolhem-se pais aleatórios com igual probabilidade da população, e geram-se 200 filhos. A recombinação da população é feita de maneira discreta, onde índices dos dois pais são escolhidos aleatoriamente para o filho, como no código abaixo:

```
function y = crossover_discrete(parent_1, parent_2)

a = randi(2, length(parent_1),1);
parent = [parent_1 , parent_2];
y = diag(parent(:,a));
```

A variável σ da mutação é alterada de maneira global intermediária, isto é, o σ filho é a média dos σ dos pais. Como no código abaixo:

```
function y = crossover_global_intermediate(s_1, s_2)
y = (s_1 + s_2) / 2;
```

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

```
function [y, s] = mutation(x, s)
global tau tau_prime epsilon
population_size = size(x,2);

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

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

Todos os filhos sofrem mutação.

Por fim, a função é avaliada para os 200 filhos e escolhem-se os 30 melhores por *comma selection*, ou seja, os pais náo sáo avaliados:

```
function y = f(x)
number_of_states = size(x,1);
y = -20 * exp(- 0.2 * sqrt(sum(x.*x)/number_of_states)) - ...
    exp(sum(cos(2*pi*x)/number_of_states)) + 20 + exp(1);
```

```
function [pop, s] = comma_selection(j, children, children_s)
global population_size
[~, jsort_index]=sort(j, 'ascend');
```

```
4 jsort_index = jsort_index(1:population_size);
5
6 pop = children(:,jsort_index);
7 s = children_s(:,jsort_index);
```

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

$$J_{\text{medio}} = 0.026$$
$$J_{\text{std}} = 0.0018$$