Primeira lista de exercícios - QUESTÃO 2

Alberto Romanhol Moreira - 2017051564

Funções de testes

Para realização do trabalho, utilizou-se de um programa desenvolvido em pyhton. O programa foi implementado para a função esfera, utilizando-se de N = 10.

Inicialmente, gera-se população com números gerados no intervalo de -5.12 a 5.12.

A partir da definição de uma função de *fitness*, são realizados operações genéticas de seleção de parentes e de mutações em cima dos filhos. Isso ocorre até se encontrar o melhor fitness ou haver limite de gerações.

Ademais, foi definido que a função fitness pode ter uma margem de até 1x10^-4, para mais ou para menos.

Bibliotecas importadas

Primeiro, importa-se as bibliotecas de terceiros que serão utilizadas no desenvolvimento do tramanho. Sendo:

- A **random** para ser geradadora de número aleatórios;

In [ ]:
```
import random as rnd
```

Geração inicial de indivíduos

É feita uma função responsável para geração inicial dos indivíduos, as soluções candidatas da primeira geração, representada pela função:

In [ ]:
```
def genotype_numbers():
    return [round(rnd.uniform(-5.12, 5.12), 5) for _ in range(n_solution)]
```

Função *fitness*

Desenvolve-se uma função de *fitness*, para avaliar a performace da função esfera candidata. Cria-se a função tanto para um vetor, as gerações, bem como para um número específico.

In [ ]:
```
def child_fitness_gen(number):
    total_sum = 0
```

```
    n = len(number)

    for i in range(n):
        for _ in range(n_solution):
            total_sum += number[i]**2

    fit = 1/(1 + abs(total_sum))

    return fit

def child_fitness(number):
    total_sum = 0

    for _ in range(n_solution):
        total_sum += number**2

    fit = 1/(1 + abs(total_sum))

    return fit
```

Mecanismo de seleção

Defini-se então, dois mecanismos de seleção.

- Um sendo uma forma de torneio. Retornando o maior candidato em uma tamanho de escolha aleatória da população.

In [ ]:
```
def selected_tournment(population):
    n = len(population)
    return bigger_fitness_genotype(population[rnd.randint(0, n-1):n])
```

- O outro sendo uma roleta russa, definida a partir da soma total de chance de fitness da população. Escolhe-se então, um candidato aleatório desta população.

In [ ]:
```
def roulette_wheel(population):
    probabilities = [probability(n) for n in population]

    total = sum(probabilities)
    pick = rnd.uniform(0, total)
    current = 0

    for i in range(len(probabilities)):
        current += probabilities[i]
        if current > pick:
            return i
```

- Função que retorna a probabilidade do indivíduo.

In [ ]:
```
def probability(chromosome):
    return child_fitness(chromosome)
```

Cria-se então, a função que será responsável por escolher o método a ser utilizado, com uma chance de 20% para o torneio.

In [ ]:
```
def pick_parent(population):
    selected_tournment_probability = 0.2

    if (selected_tournment_probability > rnd.random()):
        parent = selected_tournment(population)
    else:
        parent = population[roulette_wheel(population)]

    return parent
```

Variação genética

Define-se então, dois mecanismos de variação genética.

- Uma função, parecida com o crossover, que transforma os pais em 2 filhos, a partir de um equacionamento envolvendo os pais.

In [ ]:
```
def crossover(x, y):
    return (x+y)*x, (x+y)*y
```

- Uma função de mutação, que a partir da chance de mutação de 5%, pega uma posição aleatória no cromossomo e define um valor aleatório, no intervalo -5.12 e 5.12, para este.

In [ ]:
```
def mutate(child):
    mutation_probability = 0.05

    if (mutation_probability > rnd.random()):
        child = round(rnd.uniform(-5.12, 5.12), 5)

    return child
```

É definido então a função responsável pela geração do filho, a partir dos pais. A partir destes, escolhe-se o de maior fitness para compor a nova população.

In [ ]:
```
def pick_children(parent1, parent2):
    child1, child2 = crossover(parent1, parent2)
    child1 = mutate(child1)
    child2 = mutate(child2)

    child = bigger_fitness_genotype([child1, child2])

    return child
```

Criação de uma nova população

A partir das definições de seleção e variação genética, define-se uma função responsável pela geração de uma nova população. A partir dos métodos já citadas de `pick_parent` e `pick_children` .

In [ ]:
```

```

```
def new_generation(population):
    new_population = []
    for chrom in population:
        new_chrom = []
        for _ in range(len(chrom)):
            parent1 = pick_parent(chrom)
            parent2 = pick_parent(chrom)
            child = pick_children(parent1, parent2)
            # show_children(child)
            new_chrom.append(child)

        new_population.append(new_chrom)

    return new_population
```

Além disso, utiliza-se de uma função para exibir o cromossomo no console e o seu respectivo fitness. Função foi comentada para melhor entrega do trabalho.

In [ ]:
```
def show_children(child):
    print('chromosome = {}, fitness = {}'
        .format(str(child), fitness(child)))
```

Encontrando a melhor população

É definido uma função que irá encontrar a melhor população, que possui como critério de parada o número máximo de gerações ou quando encontra-se o *fitness* na presente geração.

In [ ]:
```
def find_best_population(population):
    generation = 0
    max_generation = 1024

    while generation < max_generation and not is_perfect_solution(population):
        generation += 1
        population = new_generation(population)
        print('generation = {}, best fitness = {}'.format(generation, (max([child_fi

    print('found a generation', generation)
    return population
```

A função de verificar se é solução perfeita `is_perfect_solution` foi criada para encontrar alguns dos elementos se encontra no itervalo de soliçao definido.

In [ ]:
```
def is_perfect_solution(population):
    for chrom in population:
        for number in chrom:
            fit = child_fitness(number)
            if (fit > (perfect_solution - limit_solution) and fit < (perfect_solutio
                return True
    return False
```

Funções auxiliares

Define-se ainda, algumas funções auxiliares, como a para retornar o indivíduo com o melhor fitness da população. Bem como o melhor número de uma geração.

In [ ]:
```
def bigger_fitness_genotype(population):
    best_fitness = 0

    for chrom in population:
        if child_fitness(chrom) > best_fitness:
            best_fitness = child_fitness(chrom)
            c = chrom

    return c
```

In [ ]:
```
def bigger_fitness_number(population):
    best_fitness = 0

    for chrom in population:
        for number in chrom:
            fit = child_fitness(number)
            if (fit > best_fitness):
                best_fitness = child_fitness(number)
                c = number

    return c
```

Programa principal

A partir das funções definidas anteriormente, é encontrado a melhor população, seguindo o critério de *fitness* ou de parada por número de gerações. E, a partir disso, é encontrado o indivíduo com maior *fitness*, não sendo necessariamente o máximo.

In [ ]:
```
print('-----------------------------------')
n_solution = 10
perfect_solution = 1/(1+0)
print('max fitness:', perfect_solution)

limit_solution = 1e-4

n_population = 10
print('n_population:', n_population)
print('-----------------------------------')

population = [ genotype_numbers() for _ in range(n_population) ]

best_pouplation = find_best_population(population)
bigger_fitness = bigger_fitness_number(best_pouplation)
```

```
-----------------------------------
max fitness: 1.0
n_population: 10
-----------------------------------
generation = 1, best fitness = 0.7755244191666016
generation = 2, best fitness = 0.9989209084059316
found a generation 2
```

A partir desse resultado, pode-se imprimir no console o resultado do programa. Apresentando os dados do problema, o tipo de solução encontrada, o valor e o fitness do indivíduo.

In [ ]:

```
print('------------------------------------')
print('n queen problem - genetic algorithm')
print('n_solution:', n_solution)
print('n_population:', n_population)
print('max fitness:', perfect_solution)
print('------------------------------------')
fit = child_fitness(bigger_fitness)
if (fit > (perfect_solution - limit_solution) and fit < (perfect_solution + limit_so
    print('found a solution by fitness')
else:
    print('no solution, stopped by generation limit')
print('found fitness:', child_fitness(bigger_fitness))
print('best_number:', bigger_fitness)
print('------------------------------------')
```

```
------------------------------------
n queen problem - genetic algorithm
n_solution: 10
n_population: 10
max fitness: 1.0
------------------------------------
found a solution by fitness
found fitness: 0.9999999819652609
best_number: 4.2467328e-05
------------------------------------
```