rastrigin

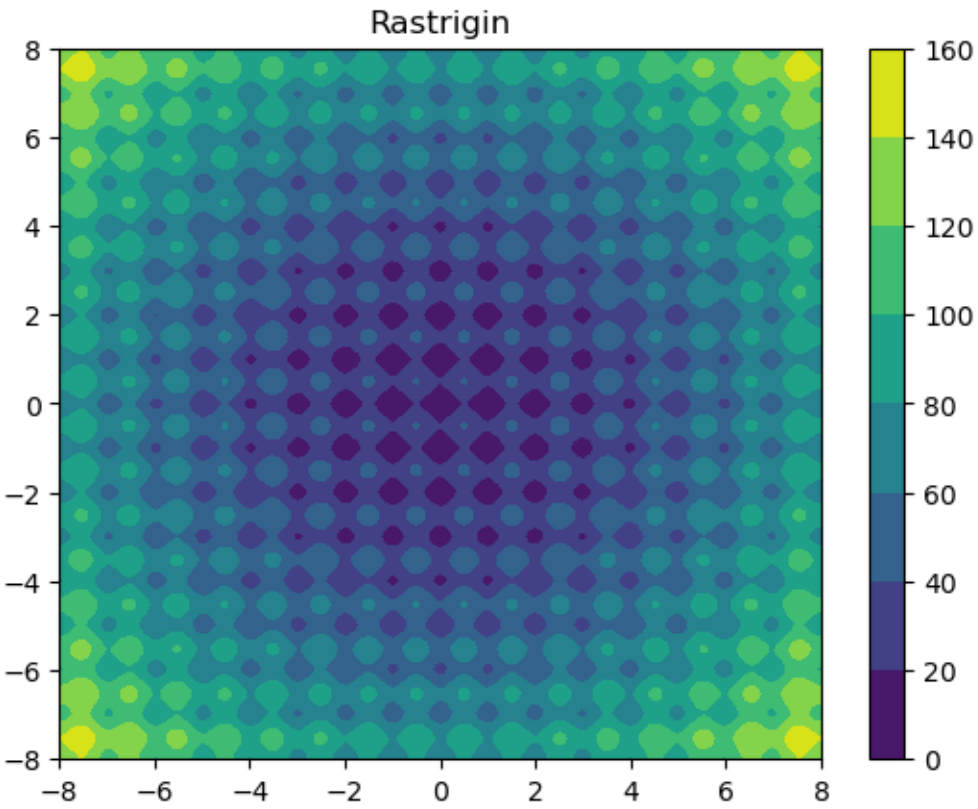
April 8, 2024

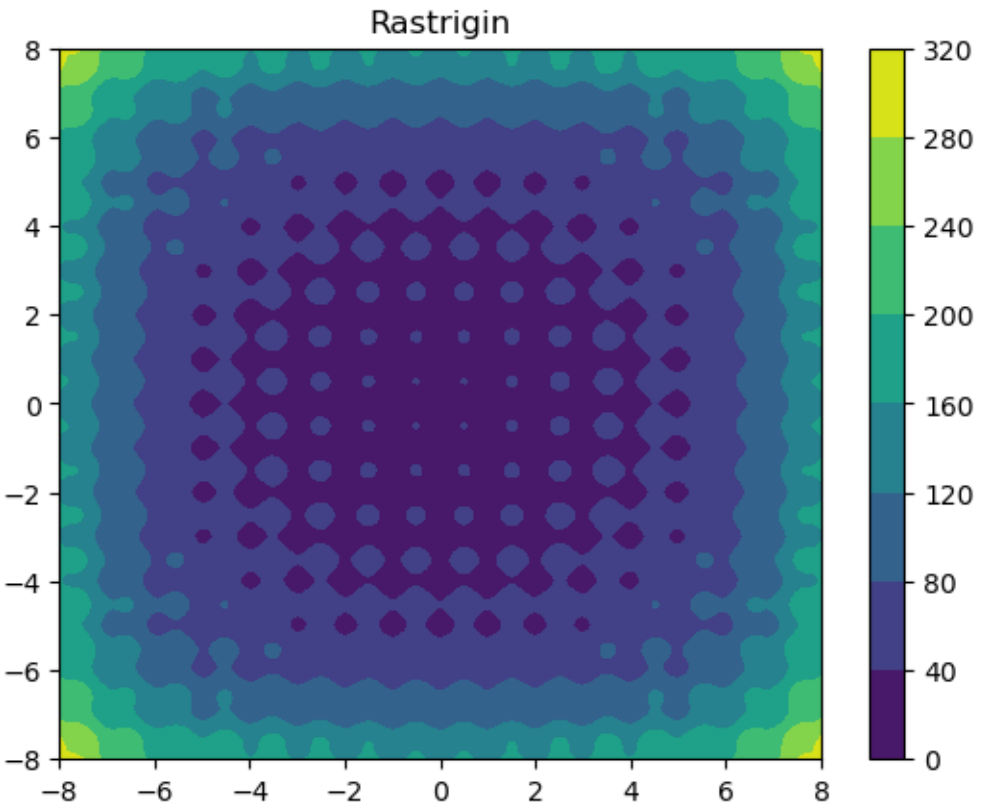
[ ]:
```
import random as rd
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from pandas import Series
from benchmark_functions import Rastrigin
```

Mostrar a função de rastringin de 2 dimensões e a mesma penalisada.

[ ]:
```
rastringin = Rastrigin(2)

points = np.linspace(-8, 8, 1000)

values = [rastringin([x, y]) for x, y in product(points, points)]

plt.figure()
plt.contourf(points, points, np.array(values).reshape(1000, 1000))
plt.colorbar()
plt.title("Rastrigin")
plt.show()

def rastringin_constrained_generator(rastringin):
    # Método utilizado: Penalização com parâmetro de penalização = 1000
    # Função de penalização 1: p * max(0, (x - 5.12))²
    # Função de penalização 2: p * max(0, (-x - 5.12))²
    # Obs: Por não ser realmente um problema de otimização convexa, iremos␣
 ↪deixar o p fixado em 10, o que
    # não é um problema por sabermos que os mínimos locais são maiores que o␣
 ↪global.
    p = 10
    def wrapper(x):
        return rastringin(x) + p * sum([max(0, (x_i - 5.12))**2 for x_i in x])␣
 ↪+ p * sum([max(0, (-x_i - 5.12))**2 for x_i in x])

    return wrapper

rastringin_constrained = rastringin_constrained_generator(rastringin)
```

```
values = [rastringin_constrained([x, y]) for x, y in product(points, points)]

plt.figure()
plt.contourf(points, points, np.array(values).reshape(1000, 1000))
plt.colorbar()
plt.title("Rastrigin")
plt.show()
```





Agora criar a função de verdade

[ ]:
```
rastringin = Rastrigin(10)

rastringin_constrained = rastringin_constrained_generator(rastringin)

def binary_to_gray(n):
    n = int(n, 2)  # Converte a string binária para um número inteiro
    n ^= (n >> 1)

    # Converte o número inteiro de volta para uma string binária
    return bin(n)[2:]

def gray_to_binary(n):
    n = int(n, 2)  # Converte a string Gray para um número inteiro
    mask = n

    # Executa operações XOR sucessivas
    while mask != 0:
        mask >>= 1
        n ^= mask
```

```
    # Converte o número inteiro de volta para uma string binária
    return bin(n)[2:]
```

Vamos utilizar duas formas de representação, para testar um pouco mais seu comportamento, a primeira é a representação binária e a segunda é a representação real.

representação binária: 10 bits para cada variável, totalizando 100 bits organizados como uma matriz 10x10.

Consideramos que [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] é -5.12 e [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] é 5.12. Assim o gap é $10.24/1024 = 0.01$

[ ]:
```
crossover_rate = 0.8
mutation_rate = 0.01 # Em média uma mutação por indivíduo
max_generations = 1000

def init_population(_mu:int = 100):
    population = []
    for i in range (_mu):
        individual = np.matrix([[rd.choice([0, 1]) for _ in range(10)] for _ in␣
 ↪range(10)])
        population.append(individual)
    return population
```

Função de aptidão

Igual à rastringin

[ ]:
```
def fitness(individual: np.matrix):
    fenotype = []
    for i in range(individual.shape[0]):
        fenotype.append(-5.12 + 10.24 * int(''.join(map(str, individual[i].
 ↪tolist()[0])), 2) / 1023)

    return rastringin_constrained(fenotype)
```

Mutação por bit flip

[ ]:
```
def mutate(solution: np.matrix, pm: float):
    for i in range(solution.shape[0]):
        for j in range(solution.shape[1]):
            if rd.random() < pm:
                solution[i, j] = 1 - solution[i, j]
    return solution
```

Seleção por roleta de 5 indivíduos

```
[ ]: def recombine(population: list[np.matrix], pc: float, selection='R') -> list[np.
     ↪matrix]:

        # Iremos utilizar um modelo com gap geracional 1/10, ou seja, criaremos 10␣
     ↪filhos por geração
        children = []
        for _ in range(10):
            # Selecionando os pais

            if selection == 'R':
                fitting_values = [1/fitness(individual) for individual in␣
     ↪population]
                fitting_values_total = sum(fitting_values)
                fitness_cumulative = Series(fitting_values).cumsum()

                # escolhendo um número aleatório na roleta
                random_number = rd.random() * fitting_values_total

                # escolhendo o primeiro pai, é o index com o primeiro número maior␣
     ↪que o número aleatório escolhido
                parent_1 = population[fitness_cumulative[fitness_cumulative >=␣
     ↪random_number].index[0]]
                parent_1_flat = parent_1.flatten()

                # O mesmo para o segundo pai
                random_number = rd.random() * fitting_values_total
                parent_2 = population[fitness_cumulative[fitness_cumulative >=␣
     ↪random_number].index[0]]
                parent_2_flat = parent_2.flatten()

            elif selection == 'T':
                random_possible_parents = [population[i] for i in rd.
     ↪sample(range(0,len(population)), 30)]

                # Melhores duas possíveis soluções para pais
                best_possible_parents_indexes = np.
     ↪argpartition([fitness(individual) for individual in random_possible_parents]␣
     ↪,2)[:2]
                parent_1 = random_possible_parents[best_possible_parents_indexes[0]]
                parent_1_flat = parent_1.flatten()
                parent_2 = random_possible_parents[best_possible_parents_indexes[1]]
                parent_2_flat = parent_2.flatten()

            if rd.random() > pc:
                children.append(parent_1)
                children.append(parent_2)
```

```
            continue

        # excolhendo o ponto de corte, a recombinação será por 1 ponto de corte␣
↪avaliando todas as variáveis em conjunto
        cut_point = rd.randint(0, 100)
        child_1 = np.concatenate((parent_1_flat[:cut_point],␣
↪parent_2_flat[cut_point:]), axis=0).reshape(10, 10)
        child_2 = np.concatenate((parent_2_flat[:cut_point],␣
↪parent_1_flat[cut_point:]), axis=0).reshape(10, 10)

        children.append(child_1)
        children.append(child_2)

    return children
```

Seleção de candidatos

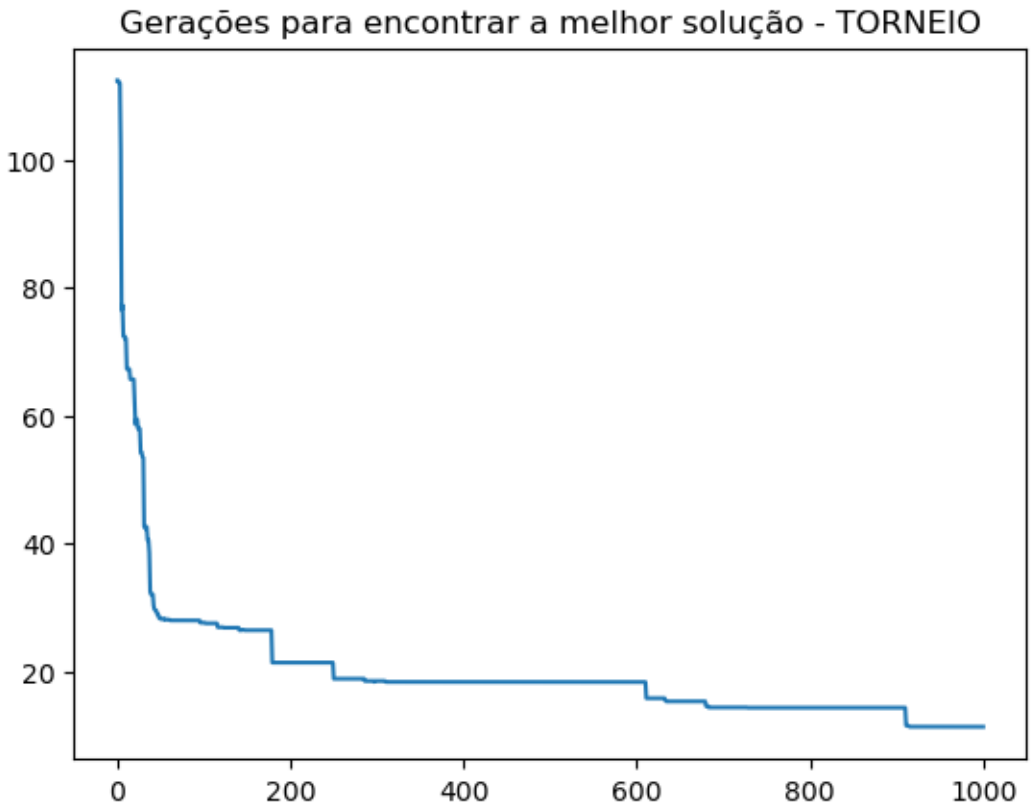
[ ]:
```
def select_new_population(pop, children):
    # Retirando piores soluções, população constante em 100

    for child in children:
        pop.append(child)

    elements_indexes_to_pop = np.argpartition([fitness(sol) for sol in pop],␣
↪-len(children))[-len(children):]
    # print(elements_indexes_to_pop, [fitness(element) for element in [pop[i]␣
↪for i in elements_indexes_to_pop]])

    for index in sorted(elements_indexes_to_pop, reverse=True):
        del pop[index]

    return pop
```

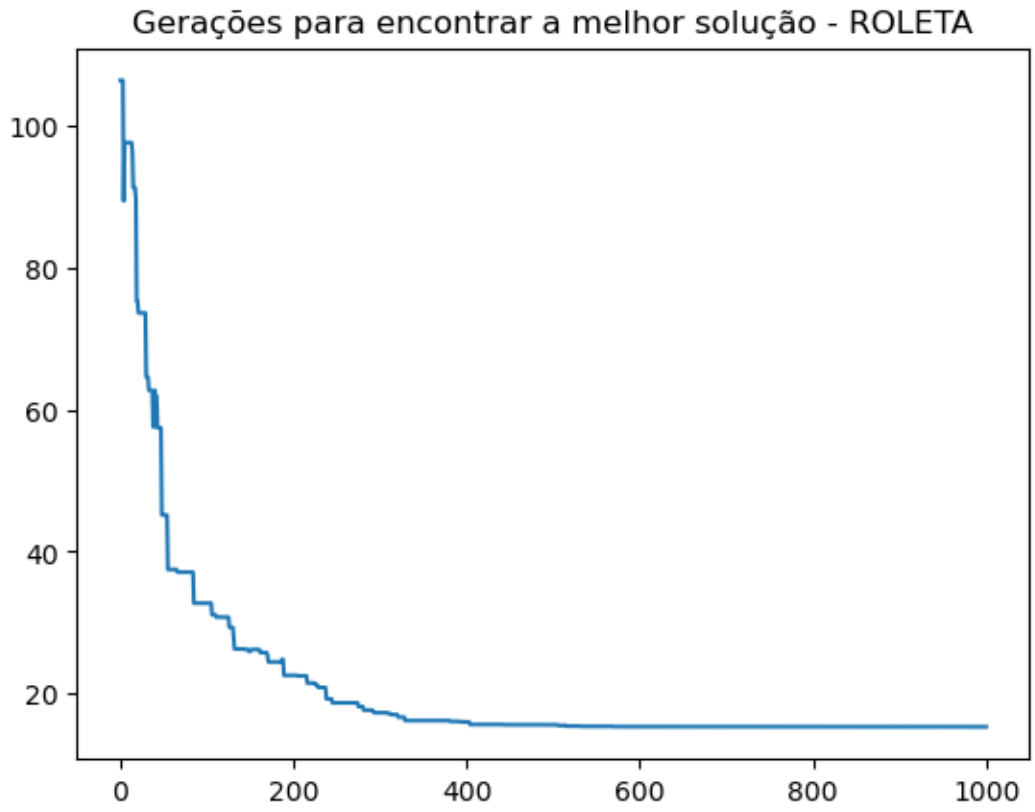
[ ]:
```
def find_best_solution(_mu, mutation_rate, selection='R'):
    pop = init_population(_mu)

    current_generation = 0
    fitness_pop_vectors = []

    while True:
        fitness_pop = [fitness(sol) for sol in pop]
        fitness_pop_vectors.append(min(fitness_pop))

        if 0 in fitness_pop or current_generation >= max_generations:
            break
        # print(current_generation, min(fitness_pop), np.argmin(fitness_pop))
```

```
        children = recombine(pop, crossover_rate, selection)

        for i, child in enumerate(children):
            child = mutate(child, mutation_rate)
            children[i] = child

        pop = select_new_population(pop, children)

        current_generation += 1

    best_solution = pop[np.argmin(fitness_pop)]

    best_solution_fitness = min(fitness_pop)

    return best_solution, best_solution_fitness, current_generation,␣
↪fitness_pop_vectors
```

Teste entre roleta e torneio

[ ]:
```
_, _, current_generation, fitness_pop_vectors = find_best_solution(100,␣
↪mutation_rate=0.01, selection='T')

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução - TORNEIO")
plt.show()

_, _, current_generation, fitness_pop_vectors = find_best_solution(100,␣
↪mutation_rate=0.01, selection='R')

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução - ROLETA")
plt.show()
```

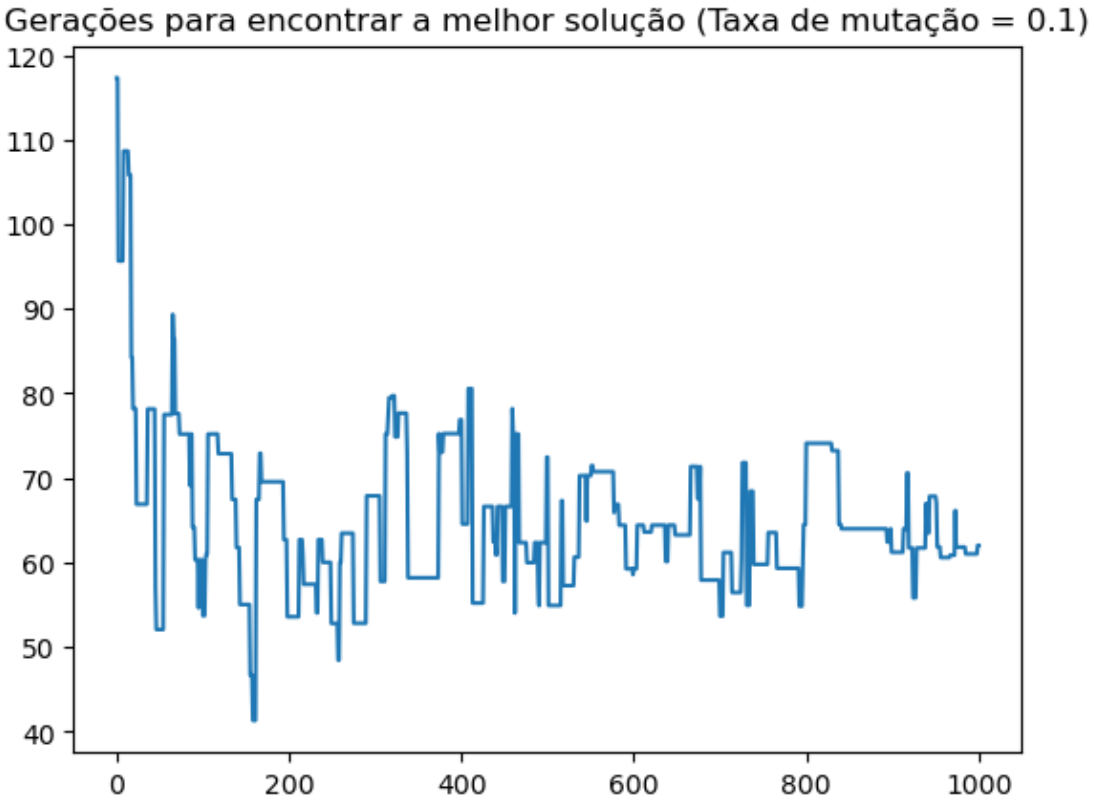

Gerações para encontrar a melhor solução - TORNEIO
100
80
60
40
20
0
200
400
600
800
1000

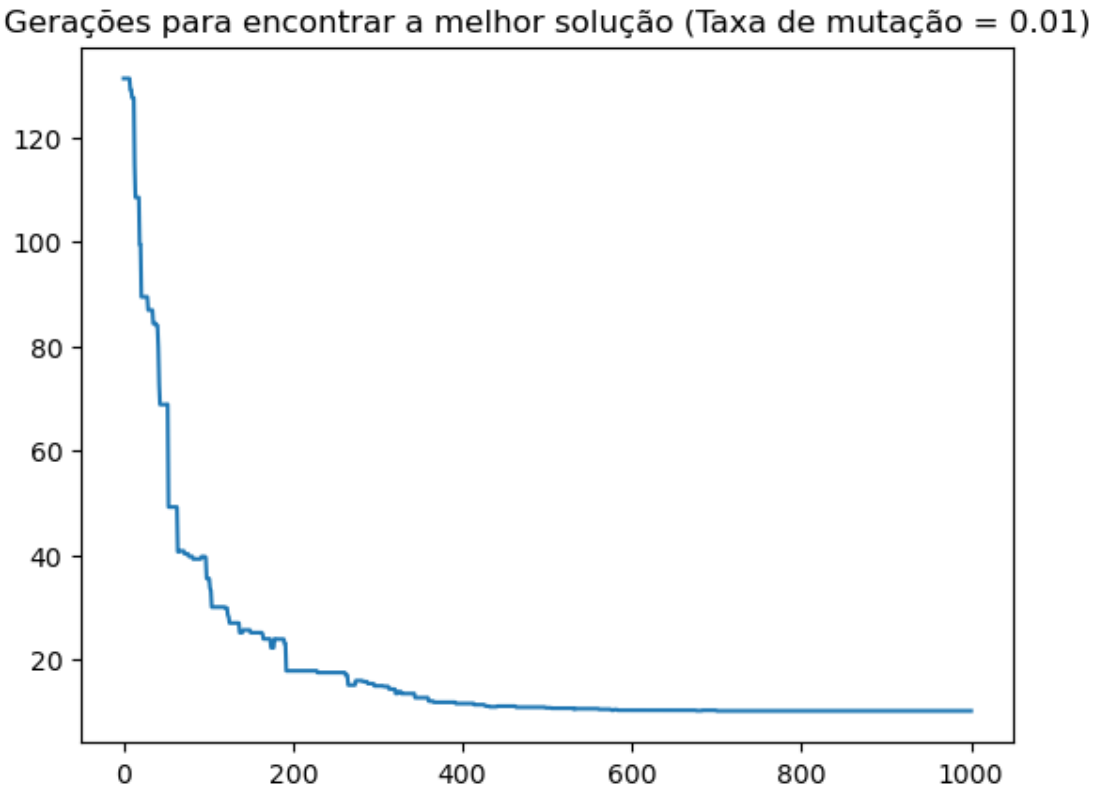


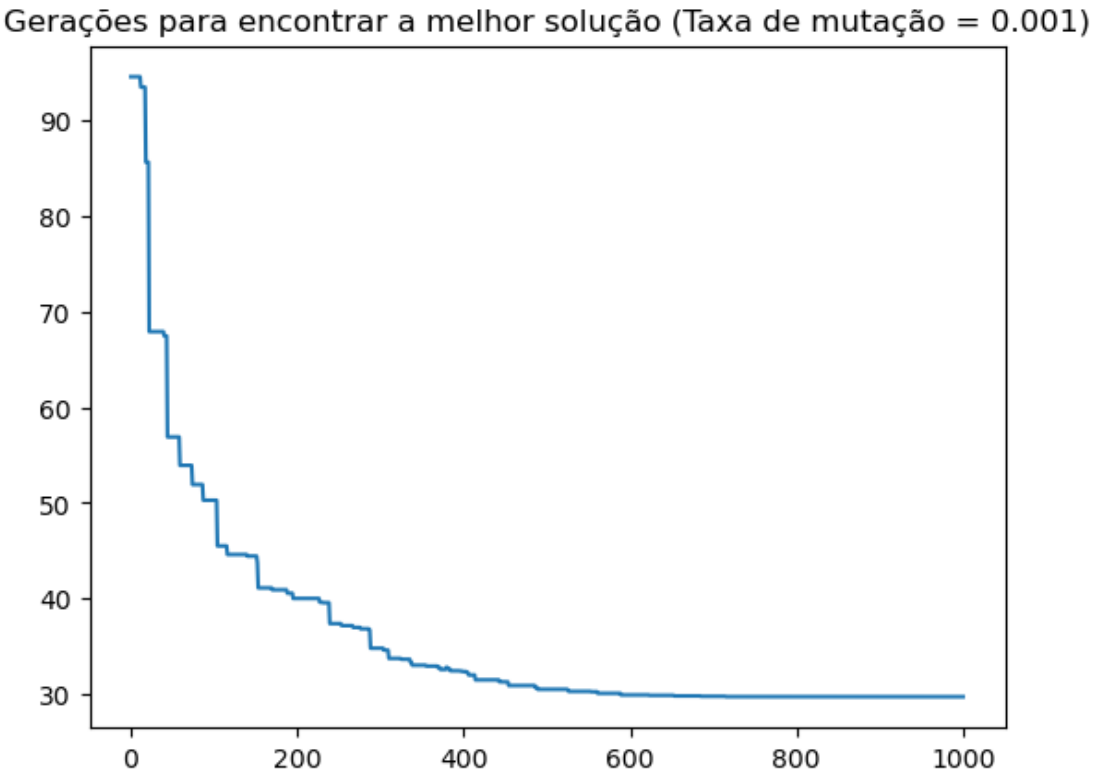
Grid search para descobrir os melhores hiperparâmetros. Primeiro para Mutação

[ ]:
```
_, _, current_generation, fitness_pop_vectors = find_best_solution(100,␣
 ↪mutation_rate=0.1)

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução (Taxa de mutação = 0.1)")
plt.show()

_, _, current_generation, fitness_pop_vectors = find_best_solution(100,␣
 ↪mutation_rate=0.01)

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução (Taxa de mutação = 0.01)")
plt.show()

_, _, current_generation, fitness_pop_vectors = find_best_solution(100,␣
 ↪mutation_rate=0.001)
```

```
plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução (Taxa de mutação = 0.001)")
plt.show()
```

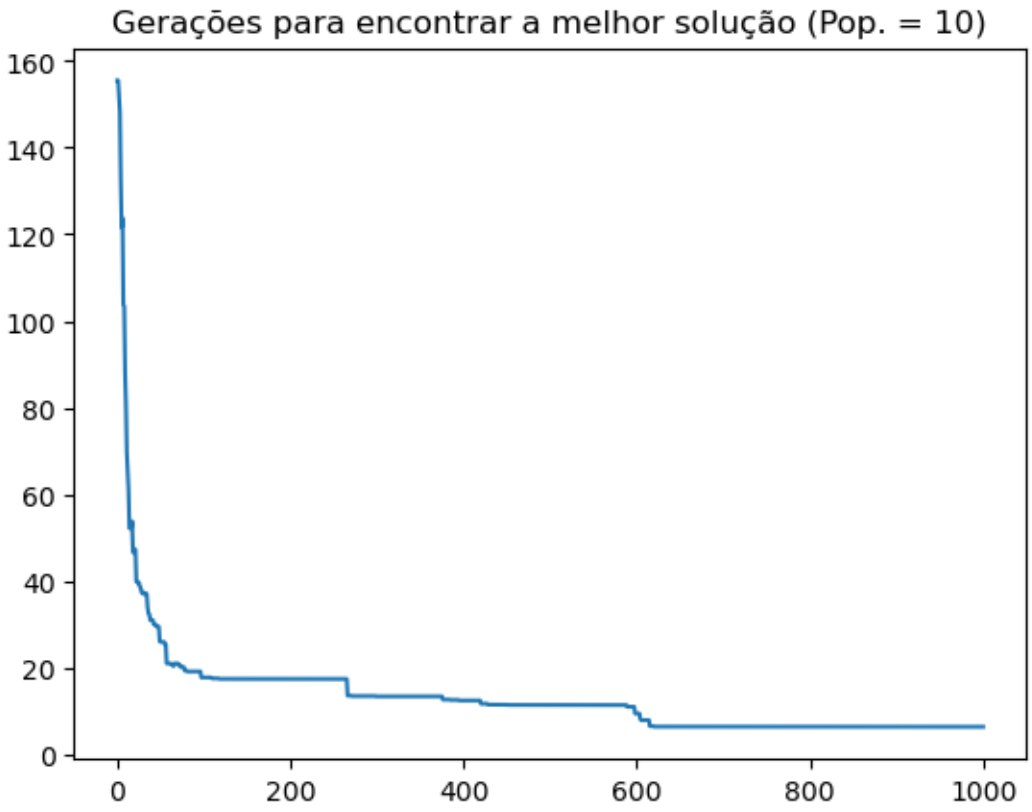


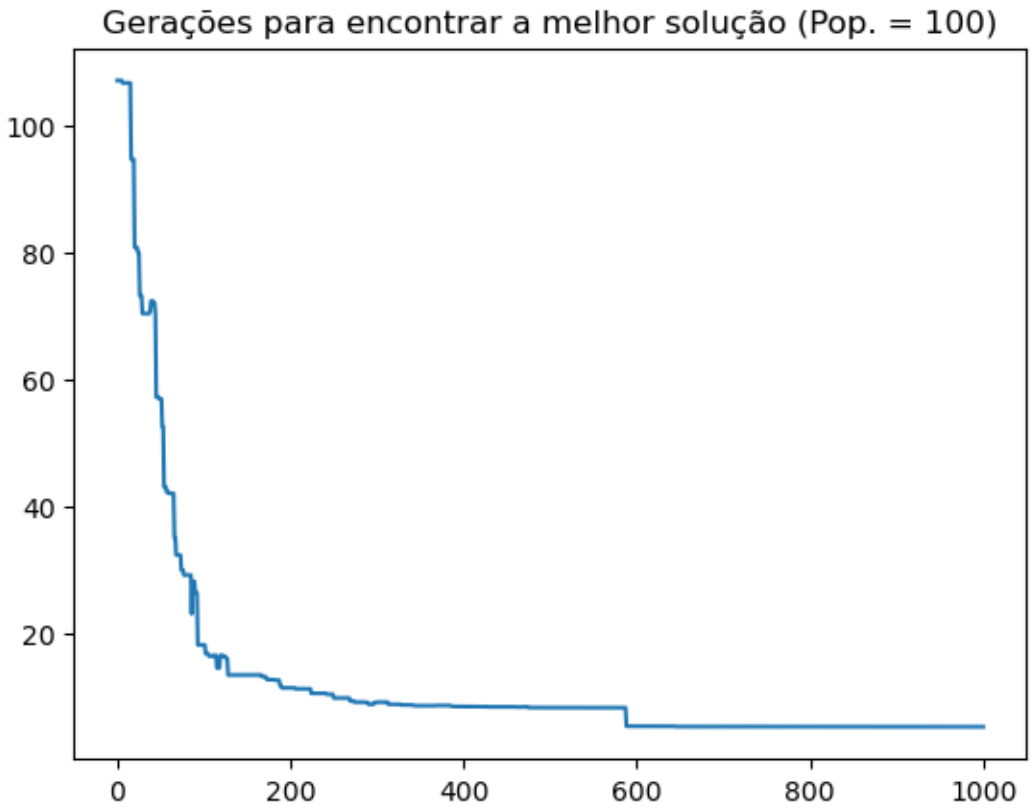

Gerações para encontrar a melhor solução (Taxa de mutação = 0.01)
120
100
80
60
40
20
0
200
400
600
800
1000


Gerações para encontrar a melhor solução (Taxa de mutação = 0.001)
90
80
70
60
50
40
30
0
200
400
600
800
1000

Descobrindo o tamanho ideal da população

[ ]:
```
analysis = dict()

_, _, current_generation, fitness_pop_vectors = find_best_solution(10,␣
 ↪mutation_rate=0.01)

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução (Pop. = 10)")
plt.show()

_, _, current_generation, fitness_pop_vectors = find_best_solution(100,␣
 ↪mutation_rate=0.01)

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução (Pop. = 100)")
plt.show()

_, _, current_generation, fitness_pop_vectors = find_best_solution(1000,␣
 ↪mutation_rate=0.01)

plt.figure()
plt.plot(fitness_pop_vectors)
plt.title("Gerações para encontrar a melhor solução (Pop. = 1000)")
plt.show()
```


Gerações para encontrar a melhor solução (Pop. = 10)
160
140
120
100
80
60
40
20
0
0
200
400
600
800
1000



Escolhi roleta com 100 indivíduos e mutação de 0.01

[ ]:

[ ]:
```
0    1000
dtype: int64
```