پیاده سازی و بررسی Genetics Algorithm

درس رایانش تکاملی – مدرس: دکتر روحانی

نویسنده: آریا عربان

هدف و راه حل

در این تمرین خواسته شده که الگوریتم GA پیاده سازی شود، تا از آن جهت پیدا کردن مقادیر بهینه چندین تابع تست استفاده شود.

برای انجام این کار، ده تابع تست در کد پیاده سازی شده است(توابع تست F1 الی F10 که در اسلایدهای 38 و 39 چپتر3 درس آمده‌اند)، و الگوریتم GA را بر روی هر کدام از این توابع 10 دفعه اجرا می‌کنیم و بررسی می‌کنیم که تغییر پارامترهای الگوریتم (تعداد جمعیت, احتمال تقاطع و احتمال جهش) به چه میزان بر روی نتیجه اثر می‌گذارد.

برای پیاده‌سازی الگوریتم GA، دقت می‌شود که هر عضوی از جمعیت، خود شامل N عدد است. هر کدام از این N عدد جهت انجام اعمال جهش و تقاطع به صورت باینری در می‌آیند. برای انجام تقاطع دو والد انتخاب شده، هر دو عضو iام این والدین به صورت یک به یک با احتمال مشخص شده‌ای، با یکدیگر این عمل را انجام می‌دهند. برای جهش نیز، هر بیت در فرزند ایجاد شده، به یک احتمال پایینی ممکن است تغییر کند.

شرح نتایج

در ابتدا، با استفاده از یکی از تابع های هدف، مقادیر بهینه برای پارامترها را تعیین می‌کنیم، تا برای مراحل بعدی از آنها استفاده شود. از تابع هدف f5 برای این کار استفاده می‌کنیم.

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \quad \text{where} \quad \mathbf{x} = (x_1, \ldots, x_N) \in \mathbb{R}^N$$

ابتدا با در نظر گرفتن مقدار ثابت احتمال تقاطع 0.9 و احتمال جهش 0.1، تابع فوق را برای مقادیر مختلف جمعیت بررسی می‌کنیم (N=2):

```
Population_size: 50 --- Objective function = 0.126
Population_size: 100 --- Objective function = 0.082
Population_size: 250 --- Objective function = 0.000102
Population_size: 500 --- Objective function = 0.000950
```

بنابرین بنظر می‌آید که مقدار مناسبی برای جمعیت برابر 250 می‌باشد.

در ادامه، با در نظر گرفتن عدد 250 برای جمعیت، و مقدار ثابت احتمال جهش برابر 0.1، تابع را برای مقادیر مختلف احتمال تقاطع بررسی می‌کنیم(N=5):

```
p_crossover: 0.2 --- Objective function = 3.746
p_crossover: 0.5 --- Objective function = 0.749
p_crossover: 0.7 --- Objective function = 1.121
p_crossover: 0.9 --- Objective function = 2.361
```

بنظر می‌آید که مقدار مناسبی برای احتمال تقاطع برابر 0.5 (یا همان 50%) می‌باشد.

در نهایت، با در نظر گرفتن مقدار 250 برای جمعیت، و احتمال تقاطع برابر 0.5، تابع را برای مقادیر مختلف احتمال جهش بررسی می‌کنیم(N=5):

```
p_mutation: 0.01 --- Objective function = 12.814
p_mutation: 0.05 --- Objective function = 3.122
p_mutation: 0.1 --- Objective function = 0.970
p_mutation: 0.2 --- Objective function = 3.573
p_mutation: 0.5 --- Objective function = 2.224
```

بنظر می‌آید که مقدار مناسبی برای احتمال جهش برابر 0.1 (یا همان 10%) می‌باشد.

با مشخص شدن مقادیر مناسب برای پارامترها، حالا به بررسی نتایج حاصل از اعمال الگوریتم GA بر روی هرکدام از توابع تست به ازاء N=10, 30, 50 پرداخته می‌شود. دقت می‌شود که در حال 10 دفعه اجرا گرفته می‌شود، و نهایتا جواب نهایی را میانگین جواب‌ها حساب می‌کنیم.

$F_1(X)$:

```
FUNCTION 1
N = 10
  best solution:
 [ 0.08344622 -0.43147048 -0.08073674 -0.03019797  0.05697899  0.0455435
 -0.30639151 -0.19769028  0.57162984  0.1156904 ]

 Objective function:
 0.678983352909146

N = 30
  best solution:
 [ 5.42352366 -3.40516067  2.16214087 -6.27204927  4.82507983 -2.63556515
 -8.96537273 -7.5547197  -2.26587636  4.55172015  9.37672084  6.75217415
 -0.35344287 -0.83393359 -8.44766806 -7.13462413 -1.57549978 10.9366055
 -2.4526443   1.00789315 -1.19765816  6.28963133 -2.71107194 11.02828599
  8.14135084  4.14536293 -2.13109521 10.46921151  5.95387752 -5.98905876]

 Objective function:
 1103.1746743610659


N = 50
 best solution:
 [-25.52236492   5.58848714  12.1124627  -21.54501795  -2.92514561
  -6.74700327 -13.32656093  -8.58735762  13.68808517  -2.5376805
  21.1254823  -20.17462719  -2.5138904   14.57566637  12.65745595
  -0.54489626  11.71816492 -15.59636656  21.63485319  14.99347573
  -9.37219661  -9.84474234  10.90957845   4.12855783  -7.16766537
  12.80834311  -5.35796891   5.27387662  14.9222878    2.06104568
  -2.73235368 -19.91211053  -1.55774292  -0.60075414  13.19361727
   5.14260723  -6.89716003 -21.57839518 -12.63963247 -12.97102016
  -6.11686505 -18.13967604   3.11554984  -2.90990044 -12.88842027
   3.32830677  -5.60274449  -2.24362545  14.81430063  12.75474436]

 Objective function:
 7347.318216989156
```

در ادامه، خروجی مختصرتر نمایش داده می‌شود.

$F_2(X)$:

```
FUNCTION 2

N = 10
 Objective function:
 0.2255752155932622 7

N = 30
 Objective function:
 28.231751032948 23

N = 50
 Objective function:
 71.213530284840 8
```

$F_3(X)$:

```
FUNCTION 3

N = 10
 Objective function:
 81.272285599259 9

N = 30
 Objective function:
 10936.55293556925 4

N = 50
 Objective function:
 69319.70642248663
```

:$F_4(X)$

```
FUNCTION 4

N = 10
 Objective function:
 0.3562206498019568 3

N = 30
 Objective function:
 9.867661086076305

N = 50
 Objective function:
 21.541510744968775
```

:$F_5(X)$

```
FUNCTION 5

N = 10
 Objective function:
 92.47266244461468

N = 30
 Objective function:
 184378.30359984218

N = 50
 Objective function:
 3393623.110436579
```

:$F_6(X)$

```
FUNCTION 6

N = 10
 Objective function:
 0.18571897578745195

N = 30
 The best solution found:
 Objective function:
 785.9087115108465

N = 50
 Objective function:
 8434.318051250813
```

:$F_7(X)$

```
 FUNCTION 7

 N = 10
  Objective function:
  1.979189086383835

 N = 30
  Objective function:
  12.966424159303942

 N = 50
  Objective function:
  30.72731471490625
```

:$F_8(X)$

```
FUNCTION 8

N = 10
 The best solution found:
 Objective function:
 -4189.768586540475

N = 30
 Objective function:
 -11782.607017793875

N = 50
 Objective function:
 -15454.292671093135
```

:$F_9(X)$

```
 FUNCTION 9

 N = 10
  Objective function:
  0.005135861763180927

 N = 30
  Objective function:
  40.99081764695947

 N = 50
  Objective function:
  160.1168315123195
```

$F_{10}(X)$:

```
FUNCTION 10

N = 10
 Objective function:
 1.7298471087925695

N = 30
 Objective function:
 21.21930928415785

N = 50
 Objective function:
 21.71658383180345
```

در تمامی مواردی که بالا بررسی شده‌اند، تعداد Iterationها را برابر 1000 هستند.

مشاهده می‌شود که در اکثر این موارد، هرچه N بزرگتر باشد، گرفتن جواب بهینه سختتر خواهد بود. علت آن است که الگوریتم باید تعداد بیشتری عدد را بهینه‌سازی کند.

البته این قانون، در برخی از موارد نیز دارای استثنا می‌باشد. مثلا در تابع $F_8(X)$، بنظر می‌آید هرچه ابعاد بیشتر شود، جواب بهینه‌تری بدست. دلیل آن است که این تابع، ابتدا هر عدد را به حالت منفی در می‌آورد، و سپس این اعداد را باهم جمع می‌کند.

در قسمت بعدی، سه مسئله که در دنیا واقعی می‌توان به کمک الگوریتم GA آنها را حل کرد را بررسی می‌کنیم. از کاربرد های این الگوریتم، می‌توان به موارد زیر اشاره کرد:

1) پیدا کردن استراترژی های سرمایه‌گذاری مناسب در بازار بورس: این الگوریتم می‌تواند با بررسی استراترژی‌های مختلف سرمایه‌گذاری، استراترژی‌هایی را از خود ابداع کند که منجر به سود بیشتر برای مشتریان شود. امروزه بخش زیادی از سیستم‌های سرمایه‌گذاری خودکار با کمک GA تولید شده‌اند.

2) تشخیص watermark در فایل صوتی: Audio Watermark یک شناسه الکترونیکی منحصر به فرد تعبیه شده در یک سیگنال صوتی است، که معمولاً برای شناسایی مالکیت حق چاپ استفاده می شود. این شناسه معمولا به روشی به نام spectrum audio watermarking (SSW) توسط ناشر بر روی فایل صوتی قرار می‌گیرد، و تلاش برای حذف آن منجر به خرابی در فایل صوتی می‌شود. SSW با استفاده سکانس pseudonoise (PN) بر روی فایل صوتی قرار می‌گیرد، و در حالت عادی کاربری که به این فایل صوتی دسترسی دارد به این اطلاعات دسترسی ندارد، اما ممکن است کاربر بتواند به این اطلاعات مخفی شده در حالتی دسترسی پیدا کند که از GA جهت پیدا کردن سکانس PN در فایل صوتی استفاده کند.

3) مسیریابی سفر، ترافیک، و محموله: یکی از کاربرد های رایج GA، حل مسئله Traveling Salesman Problem (TSP) می‌باشد. حل این مسئله، می تواند برای پیدا کردن کارآمدترین مسیرها برای برنامه ریزان سفر، مسیریاب‌های ترافیکی و حتی شرکت‌های حمل و نقل استفاده شود. پیدا کردن کوتاه‌ترین مسیر سفر، پیدا کردن زمان‌بندی برای جلوگیری از ترافیک و ساعات شلوغی و پیدا کردن کارآمدترین استفاده از وسیله نقلیه برای محموله، همگی نمونه کاربردهایی از حل این مسئله هستند.

از بین این کاربردهای واقعی GA، مسئله TSP برای بدست آوردن کارامدترین مسیریابی بین‌شهری پیاده سازی شده است.

در پیاده سازی، هر عضوی از جمعیت شامل یک آرایه است با طول تعداد کل شهرها، و ترتیب این آرایه برابر ترتیب عبور از شهرها برای آن عضو به‌خصوص می‌باشد. مثلا در حالتی که 4 شهر داریم، ممکن است یکی از اعضا جمعیت مقداری برابر [0,2,1,3] داشته باشد.

از آنجایی که مسئله TSP، یک مسئله جایگشت است، پس برای تقاطع از روش OneMax Crossover استفاده می‌شود.

به عنوان یک مسئله تست، فرض می‌کنیم که 15 شهر با مختصات زیر داریم

```
cityCoordinates = [[5, 80], [124, 31], [46, 54], [86, 148], [21, 8],
                   [134, 72], [49, 126], [36, 34], [26, 49], [141, 6],
                   [124, 122], [80, 92], [70, 69], [76, 133], [23, 65]]
```

و می‌دانیم که جواب بهینه تقریبا برابر 550 می‌باشد.

با اجرا گرفتن از کد با مقدار جمعیت اولیه 100، خواهیم داشت:

```
Generation: 1
    Average Fitness: 1204.64
Best Fitness: 915.96


Generation: 2
    Average Fitness: 1180.23
Best Fitness: 838.98

...

Generation: 72
    Average Fitness: 745.51
Best Fitness: 632.77

...

Generation: 96
    Average Fitness: 734.27
Best Fitness: 587.57

...

Generation: 103
    Average Fitness: 731.43
Best Fitness: 587.57

...

Generation: 207
    Average Fitness: 711.79
Best Fitness: 571.91
```

در قسمت آخر، به تقسیم یک مسئله واحد GA با دامنه مشخص، به دوتا زیر مسئله که دامنه هر کدام نصف دامنه اصلی هستند میپردازیم و این دو حالت را باهم مقایسه می‌کنیم.

با انجام این کار بر روی توابع تست F1 الی F5، متوجه می‌شویم که در تمامی این حالت‌ها، مدل اصلی با دامنه کامل از هر دو مدل با نصف دامنه بهتر عمل می‌کند. به عنوان مثال، در F1 داریم:

```
FULL DOMAIN
 The best solution found:
 [-9.99702425e-05 -2.08058509e-03  2.42942816e-04]

 Objective function:
 4.397849563755008e-06

-----
HALF 1 DOMAIN
The best solution found:
 [-0.00087694 -0.08127667 -0.02548281]

 Objective function:
 0.007256040415164199
------
HALF 2 DOMAIN
 The best solution found:
 [0.03516917 0.00507766 0.14075513]

 Objective function:
 0.021074659423206358
```

هرچند هر دو جواب با دامنه های تقسیم شده قابل قبولند، اما مشخص است که تابع اصلی به جواب بهینه بسیار نزدیکتر است.

ضمیمه (کد) :

پیاده سازی توابع هدف:

```
def f1(v):
    total = 0
    for i in range(len(v)):
        xi = v[i] ** 2
        total = total + xi
    return np.abs(total)


def f2(v):
    return np.sum(np.abs(v)) + np.prod(np.abs(v))


def f3(v):
    total = 0
    for i in range(len(v)):
        total = total + (np.sum(v[:(i+1)]) ** 2)
    return total


def f4(v):
    return np.max(np.abs(v))


def f5(v):
    ### ROSENBROCK ###
    total = 0
    for i in range(len(v) - 1):
        xi = v[i]
        x_next = v[i + 1]
        new = 100 * (x_next - xi ** 2) ** 2 + (xi - 1) ** 2
        total = total + new
    return total
```

```python
def f6(v):
    total = 0
    for i in range(len(v)):
        total = total + ((v[i] + 0.5) ** 2)
    return total


def f7(v):
    total = 0
    for i in range(len(v)):
        total = total + (i+1) * (v[i] ** 4) + random.uniform(0, 1)
    return total


def f8(v):
    total = 0
    for i in range(len(v)):
        total = total - (v[i] * math.sin(math.sqrt(np.abs(v[i]))))
    return total


def f9(v):
    total = 0
    for i in range(len(v)):
        total = total + ((v[i] ** 2) - (10 * math.cos(2 * math.pi * v[i])) + 10)
    return total


def f10(v):
    ttl1 = 0
    ttl2 = 0
    for i in range(len(v)):
        ttl1 = ttl1 + (v[i] ** 2)
        ttl2 = ttl2 * math.cos(2 * math.pi * v[i])

    total = -20 * math.exp(-0.2 * math.sqrt(ttl1)) - math.exp((1 / len(v)) * ttl2) + 20 + math.e
    return total
```

```
funcs = {
    1: {'function_name': f1, 'bounds': [-100, 100]},
    2: {'function_name': f2, 'bounds': [-10, 10]},
    3: {'function_name': f3, 'bounds': [-100, 100]},
    4: {'function_name': f4, 'bounds': [-100, 100]},
    5: {'function_name': f5, 'bounds': [-30, 30]},
    6: {'function_name': f6, 'bounds': [-100, 100]},
    7: {'function_name': f7, 'bounds': [-1.28, 1.28]},
    8: {'function_name': f8, 'bounds': [-500, 500]},
    9: {'function_name': f9, 'bounds': [-5.12, 5.12]},
    10: {'function_name': f10, 'bounds': [-32, 32]},
    11: {'function_name': f11, 'bounds': [-600, 600]},
}
```

پیاده سازی الگوریتم GA:

```python
from numpy.random import randint
from numpy.random import rand
from GA.lib.objective_functions import funcs
import numpy as np


# decode bitstrings of one person in the population to numbers
def decode(bounds, n_bits, bitstrings):
    decoded = list()
    largest = 2 ** n_bits
    for bts in bitstrings:
        substring = bts
        # convert bitstring to a string of chars
        chars = ''.join([str(s) for s in substring])

        # convert string to integer
        integer = int(chars, 2)
        # scale integer to desired range
        value = bounds[0] + (integer / largest) * (bounds[1] - bounds[0])
        # store
        decoded.append(value)
    return decoded


# tournament selection
def tournament_selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k - 1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    # print(pop[selection_ix])
    return pop[selection_ix]
```

```python
def roulette_selection(pop, scores):
    if np.min(scores) < 0:
        scores = scores - np.min(scores)

    scores = np.max(scores) - scores
    # first random selection
    div_scores = scores / np.sum(scores)
    rnd_num = random.uniform(0, 1)
    idx = -1
    for i in range(len(scores)):
        rnd_num = rnd_num - div_scores[i]
        if (rnd_num < 0):
            idx = i
            break
    return pop[idx]


# crossover two parents to create two children
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()

    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string

        for i in range(len(c1)):
            pt = randint(1, len(p1[0]) - 2)
            # perform crossover
            c1[i] = p1[i][:pt] + p2[i][pt:]
            c2[i] = p2[i][:pt] + p1[i][pt:]

    return [c1, c2]
```

```python
def mutation(bitstrings, r_mut):
    for bts in bitstrings:
        for i in range(len(bts)):
            # check for a mutation
            if rand() < r_mut:
                # flip the bit

                bts[i] = 1 - bts[i]


# genetic algorithm
def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, n_inside_parent, r_cross, r_mut):
    # initial population of random bitstring
    pop = [[randint(0, 2, n_bits).tolist() for _ in range(n_inside_parent)] for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(decode(bounds, n_bits, pop[0]))
    # enumerate generations
    for gen in range(n_iter):
        # decode population
        decoded = [decode(bounds, n_bits, p) for p in pop]
        # evaluate all candidates in the population
        scores = [objective(d) for d in decoded]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
            # print(">%d, new best f(%s) = %f" % (gen, decoded[i], scores[i]))
        # select parents
        selected = [roulette_selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i + 1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]
```

```python
def GA_main(population_size, n_inside_parent, p_crossover, p_mutation, func_num):
    # define the total iterations
    n_iter = 1000
    # bits per variable
    n_bits = 16
    # define the population size
    n_pop = population_size
    # crossover rate
    r_cross = p_crossover
    # mutation rate
    r_mut = p_mutation
    # perform the genetic algorithm search

    n_inside_parent = n_inside_parent

    objective_func = funcs[func_num]  # g1t objective function and bound pair from the funcs dictionary, can be 1 to 10

    best, score = genetic_algorithm(objective_func['function_name'], objective_func['bounds'], n_bits, n_iter, n_pop,
                                    n_inside_parent, r_cross, r_mut)
    # print('Done!')
    decoded = decode(objective_func['bounds'], n_bits, best)
    # print('f(%s) = %f' % (decoded, score))
    return decoded, score


def main():
    for i in range(10):
        print(f"FUNCTION {i + 1}")
        for j in [10, 30, 50]:
            print(f"N = {j}")
            sols = []
            obj = []
            for k in range(10):
                best_solution, objective_function_score = GA_main(population_size=250,
                                                                  n_inside_parent=j, p_crossover=0.5,
                                                                  p_mutation=0.1, func_num=i + 1)
                obj.append(objective_function_score)
                sols.append(best_solution)
            print(f"best_solution:\n {np.mean(sols, axis=0)}")
            print(f"objective_function:\n {np.mean(obj)}")
            print("------------")


if __name__ == '__main__':
    main()
```

پیاده‌سازی GA برای حل TSP:

```python
import numpy as np
import random
import math

# from visualize import

MUTATION_RATE = 60
MUTATION_REPEAT_COUNT = 2
WEAKNESS_THRESHOLD = 850

# Begin and end point is first city
cityCoordinates = [[5, 80], [124, 31], [46, 54], [86, 148], [21, 8],
                   [134, 72], [49, 126], [36, 34], [26, 49], [141, 6],
                   [124, 122], [80, 92], [70, 69], [76, 133], [23, 65]]

citySize = len(cityCoordinates)


class Genome():
    chromosomes = []
    fitness = 9999


def CreateNewPopulation(size):
    population = []
    for x in range(size):
        newGenome = Genome()
        newGenome.chromosomes = random.sample(range(1, citySize), citySize - 1)
        newGenome.chromosomes.insert(0, 0)
        newGenome.chromosomes.append(0)
        newGenome.fitness = Evaluate(newGenome.chromosomes)
        population.append(newGenome)
    return population


# Calculate distance between two point
def distance(a, b):
    dis = math.sqrt(((a[0] - b[0]) ** 2) + ((a[1] - b[1]) ** 2))
    return np.round(dis, 2)
```

```
def Evaluate(chromosomes):
    calculatedFitness = 0
    for i in range(len(chromosomes) - 1):
        p1 = cityCoordinates[chromosomes[i]]
        p2 = cityCoordinates[chromosomes[i + 1]]
        calculatedFitness += distance(p1, p2)
    calculatedFitness = np.round(calculatedFitness, 2)
    return calculatedFitness


def findBestGenome(population):
    allFitness = [i.fitness for i in population]
    bestFitness = min(allFitness)
    return population[allFitness.index(bestFitness)]


# In K-Way tournament selection, we select K individuals
# from the population at random and select the best out
# of these to become a parent. The same process is repeated
# for selecting the next parent.
def TournamentSelection(population, k):
    selected = [population[random.randrange(0, len(population))] for i in range(k)]
    bestGenome = findBestGenome(selected)
    return bestGenome


def Reproduction(population):
    parent1 = TournamentSelection(population, 10).chromosomes
    parent2 = TournamentSelection(population, 6).chromosomes
    while parent1 == parent2:
        parent2 = TournamentSelection(population, 6).chromosomes

    return OrderOneCrossover(parent1, parent2)


# Sample:
```

```
def OrderOneCrossover(parent1, parent2):
    size = len(parent1)
    child = [-1] * size

    child[0], child[size - 1] = 0, 0

    point = random.randrange(5, size - 4)

    for i in range(point, point + 4):
        child[i] = parent1[i]
    point += 4
    point2 = point
    while child[point] in [-1, 0]:
        if child[point] != 0:
            if parent2[point2] not in child:
                child[point] = parent2[point2]
                point += 1
                if point == size:
                    point = 0
            else:
                point2 += 1
                if point2 == size:
                    point2 = 0
        else:
            point += 1
            if point == size:
                point = 0

    if random.randrange(0, 100) < MUTATION_RATE:
        child = SwapMutation(child)

    # Create new genome for child
    newGenome = Genome()
    newGenome.chromosomes = child
    newGenome.fitness = Evaluate(child)
    return newGenome
```

```python
def SwapMutation(chromo):
    for x in range(MUTATION_REPEAT_COUNT):
        p1, p2 = [random.randrange(1, len(chromo) - 1) for i in range(2)]
        while p1 == p2:
            p2 = random.randrange(1, len(chromo) - 1)
        log = chromo[p1]
        chromo[p1] = chromo[p2]
        chromo[p2] = log
    return chromo


def GeneticAlgorithm(popSize, maxGeneration):
    allBestFitness = []
    population = CreateNewPopulation(popSize)
    generation = 0
    while generation < maxGeneration:
        generation += 1

        for i in range(int(popSize / 2)):
            # Select parent, make crossover and
            # after, append in population a new child
            population.append(Reproduction(population))

        # Kill weakness person
        for genom in population:
            if genom.fitness > WEAKNESS_THRESHOLD:
                population.remove(genom)

        averageFitness = round(np.sum([genom.fitness for genom in population]) / len(population), 2)
        bestGenome = findBestGenome(population)
        print("\n" * 5)
        print("Generation: {0}\n\t Average Fitness: {1}\nBest Fitness: {2}"
              .format(generation, averageFitness,
                      bestGenome.fitness))

        allBestFitness.append(bestGenome.fitness)

    # Visualize


# plot(generation, allBestFitness, bestGenome, cityCoordinates)


if __name__ == "__main__":
    GeneticAlgorithm(popSize=100, maxGeneration=300)
```

کد برای اجرا GA با دامنه تقسیم شده:

```python
def GA_divide(population_size, n_inside_parent, p_crossover, p_mutation, func_num):
    # define the total iterations
    n_iter = 1000
    # bits per variable
    n_bits = 16
    # define the population size
    n_pop = population_size
    # crossover rate
    r_cross = p_crossover
    # mutation rate
    r_mut = p_mutation
    # perform the genetic algorithm search

    n_inside_parent = n_inside_parent

    objective_func = funcs[func_num]  # g1t objective function and bound pair from the funcs dictionary, can be 1 to 10
    hlf = split_list(objective_func['bounds'])
    best1, score1 = genetic_algorithm(objective_func['function_name'], hlf[0], n_bits, n_iter, n_pop,
                                      n_inside_parent, r_cross, r_mut)
    best2, score2 = genetic_algorithm(objective_func['function_name'], hlf[1], n_bits, n_iter, n_pop,
                                        n_inside_parent, r_cross, r_mut)

    best, score = genetic_algorithm(objective_func['function_name'], objective_func['bounds'], n_bits, n_iter, n_pop,
                                    n_inside_parent, r_cross, r_mut)
    # print('Done!')
    decoded = decode(objective_func['bounds'], n_bits, best)
    decoded1 = decode(hlf[0], n_bits, best1)
    decoded2 = decode(hlf[1], n_bits, best2)
    # print('f(%s) = %f' % (decoded, score))
    return decoded, score, decoded1, score1, decoded2,score2
```